

代 号 10701

学 号 1077190439

分类号 TP316.2

密 级 公开

U D C

编 号

题 (中、英文) 目 基于 L4 的微内核的设计与实现

Design and Implementation of a Microkernel Based on L4

作 者 姓 名 焦向 学校指导教师姓名职称 鱼滨 教授

工 程 领 域 计算机技术 企业指导教师姓名职称 张晓红 研究员

论 文 类 型 应用基础技术 提交论文日期 二〇一三年三月

西安电子科技大学 学位论文独创性声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中做了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切法律责任。

本人签名：_____ 日 期：_____

西安电子科技大学 关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。同时本人保证，毕业后结合学位论文研究课题再撰写的文章一律署名为西安电子科技大学。

（保密的论文在解密后遵守此规定）

本学位论文属于保密，在 _____ 年解密后适用本授权书。

本人签名：_____ 导师签名：_____

日 期：_____ 日 期：_____

摘要

随着软硬件的发展，嵌入式设备已成为人们日常生活必不可少的工具。嵌入式系统已经深入到社会生活的方方面面，并已不满足于单一功能，其职能越来越接近通用操作系统，这就对操作系统的精简性和安全性提出了更高的要求。近年来，微内核理论在嵌入式领域的应用与日俱增。其中，以 L4 为代表的第二代微内核理论发展尤其迅猛，不仅在学术界硕果累累，在工业界也有许多案例大量部署基于该理论设计的操作系统。本论文探讨并设计了一款基于 L4 理论的微内核，该系统基于 ARM 体系结构，实现了完整的 L4 规范，使用容器的方式隔离不同的执行环境，并探讨如何基于该微内核构建通用操作系统和在此基础上实现虚拟化。

论文首先介绍了微内核的概念和发展历程，详细阐述了 L4 微内核如何解决传统微内核的性能缺陷。然后详细描述了系统的整体架构，并对线程模型、内存管理、线程间通信、线程调度和容器分别进行了细致的分析。关于各模块的实现方式结合了代码和文字语言，深入直观的说明了主要功能和核心算法的实现细节。一步一步讲述如何从零实现一个符合 L4 规范的微内核，并对实现过程中的设计决策做出分析。

关键词：微内核 L4 操作系统 虚拟化

Abstract

With the development of software and hardware, embedded equipment has become an essential tool of people's daily life. Embedded system is not satisfied with single purpose function, but more and more close to the general operation system. This requires the operation system to be more compact and more security. In recent years, microkernel theory in the field of embedded application grows day by day. Not only in academic but also in industry, the second generation of microkernel theory, especially L4, develops rapidly. There are many successful cases based on the operating system using L4 theory. This paper discusses a kernel based on the theory of L4 microkernel, it implements all of the L4 specification and adopts container way to isolate different execution environments on ARM architecture. The ending of the paper discusses how to build a general purpose operating system based on the microkernel and the implementation of paravirtualization.

The paper introduces the concept and development of microkernel firstly and describes the problems of traditional microkernels in detail, especially performance problem. Secondly, a detailed description of the whole structure of the system follows, this part includes the thread model, memory management, communication between threads, thread scheduling and container. To illustrate the main function and the core algorithm clearly and deeply, the analysis of each module's implementation combines code and language. It shows the process of implementing a L4 microkernel step by step and every design decision during the implementation.

Keywords: Microkernel L4 Operating System Virtualization

目录

第一章 绪论.....	1
1.1 引言.....	1
1.2 研究背景和意义.....	1
1.3 论文组织结构.....	3
第二章 微内核技术概述.....	5
2.1 微内核.....	5
2.2 L4 微内核.....	6
2.2.1 L4 的起源.....	6
2.2.2 L4 微内核规范.....	6
2.2.3 L4 的实现及其现状.....	7
2.2.4 L4 的前沿.....	9
2.2.5 重要概念和技术.....	10
第三章 总体设计.....	13
3.1 设计原则和特点.....	13
3.2 核心概念.....	14
3.3 Nros 的架构.....	16
3.3.1 平台相关层.....	17
3.3.2 通用服务层.....	17
3.3.3 粘合层.....	18
3.3.4 API 层.....	18
3.3.5 支持库.....	19
第四章 实现.....	21
4.1 线程和调度.....	21
4.1.1 概述.....	21
4.1.2 线程的结构.....	21
4.1.3 线程的生命周期.....	23
4.1.4 线程上下文转换.....	25
4.1.5 线程调度.....	27
4.2 线程间通信（IPC）.....	30
4.2.1 概述.....	30
4.2.2 消息类型.....	30

4.2.3 消息拷贝方法和效率.....	32
4.3.4 IPC 实现	34
4.3 容器.....	38
4.3.1 容器概述.....	38
4.3.2 容器的结构.....	39
4.3.3 容器的静态结构.....	40
4.4 内存管理.....	42
4.4.1 内存管理概述.....	42
4.4.2 ARM 内存管理单元.....	42
4.4.3 虚拟内存模型.....	43
4.4.4 虚拟内存接口.....	44
4.4.5 地址空间 (Address Space)	46
4.4.6 地址空间的使用举例 (使用地址和线程接口实现 fork()调用)	
.....	47
第五章 构建与测试.....	49
5.1 构建.....	49
5.2 测试运行 Nros.....	53
5.2.1 运行 Nros.....	53
5.2.2 测试 Nros 实例.....	56
5.2.3 开发新容器.....	58
第六章 总结与后续工作.....	59
6.1 总结.....	59
6.2 后续工作.....	59
6.2.1 虚拟化 Linux	59
6.2.2 兼容 POSIX	60
致谢.....	61
参考文献.....	63

第一章 绪论

1.1 引言

微内核理论已存在超过三十年，L4 微内核理论也已经诞生了近二十年，然而主流操作系统似乎一直排斥微内核设计，尽管意识到微内核的优势，也仅愿意做到部分的采用。关于微内核和宏内核孰优孰劣的讨论从未真正停止过。时至今日 Linux 之父 Linus Torvalds 仍然坚持微内核是失败的设计；相比之下市场份额相对较小的 BSD 分支和封闭的 Windows 系统却更愿意接纳微内核理论。苹果公司的 Mac OS X 和 iOS 操作系统基于 BSD 系统，部分采用微内核设计；从 FreeBSD 衍生出来的 DragonFlyBSD 创造的 Light Weight Kernel Thread 概念，也采用了微内核的思想；微软公司的 Windows NT 系统也借鉴了微内核设计。然而不论是 Windows 内核还是 UNIX 内核，其微内核思想都受 Mach 操作系统的影响，Mach 是一个研究性质的操作系统，由于它最终没有做出一个可以实用的系统，却在内核开发者社区里有着巨大的影响力，使得人们以为微内核理论已经发展到了尽头。然而事实并非如此，在 Mach 之外，关于微内核究竟该如何实现的研究一直在进行。L4 微内核理论诞生于二十世纪九十年代，此后蓬勃发展二十年，从研究机构 and 高等院校走到工业界，并成功实现商业化。2010 年 3 月 Open Kernel Labs 宣布其 OKL4 已部署在超过十亿台移动设备。说明 L4 微内核具有巨大的商业价值。可以预见，在未来随着硬件性能的提升和成本的降低，微内核将会有更大的用武之地。

本论文实现了一款基于 L4 理论的微内核，通过分析和代码解读的方式探讨 L4 微内核实现的方方面面。该微内核取名为 Nros，是一款基于 ARM 平台的小型操作系统。之所以选择 ARM 平台，是因为这个平台是目前嵌入式领域最流行、使用最广泛的平台，具有一定的代表性。其次是因为针对这个平台存在大量成熟的开发和测试工具，为实现提供了便利性。

1.2 研究背景和意义

通用操作系统非常庞大，代码量已经达到百万级甚至千万级，一些特殊用途的专有系统的代码量也颇为惊人。庞大代码库使系统非常难于升级和维护，除了少数几个最初设计系统的人了解系统的架构以外，没有人真正了解整个系统。微内核在设计上将更多的内核组件放到用户空间，可以有效降低系统的复杂性和体积。尤其在嵌入式领域，通常要求系统体积小，功能单纯且内存资源占用率低。

这些需求使用通用操作系统显得过于臃肿，嵌入式系统常常无法从通用操作系统如 Linux 的一些高级算法和数据结构获益，反而容易造成资源浪费。比如一个图像处理系统，它需要执行的任务仅仅是处理放在内存某个区域的图像数据，然后将结果通过 USB 传输出去，在此过程中，该图像处理板还要处理中断事件。这样一个系统如果基于 Linux 来开发，就显得有些杀鸡用牛刀，而如果在裸机上实现，又重复了许多无谓的劳动，这时最好的办法就是借助微内核的优势，在微内核基础之上，编写所需要的服务程序。

嵌入式程序还需要较高的安全性，微内核在安全性方面也要优于通用内核。由于更小的内核设计，使得对于安全的保障更容易进行，甚至进行形式化证明。例如，seL4 微内核的研究人员已经使用形式化的方式证明了该内核永远不会崩溃，且永远不会执行不安全的操作。这样高度安全的内核，使得其在军工和医疗器械等需要高度安全性的领域有极大的发展空间。

微内核研究可以分为两代：第一代要以 Minix 和 Mach 为代表。Minix 由 Andrew S. Tanenbaum 创造，Mach 是卡内基梅隆大学的研究项目，是 UNIX 的微内核设计。它们的思想都是将原本存在于内核空间的子系统移到用户空间，比如设备驱动和文件系统。Minix 主要作为教材的配套代码，用于学习操作系统的概念和实现细节，并不求快速发展和演变。而 Mach 作为一个研究项目，却深刻影响了以后操作系统的演化。Mach 的问题是，相比原来的单体内核，它的性能不尽人意。因此很多人投入了新型微内核的研究，这就是第二代微内核。德国 Bielefeld 大学的 L4 就是这次研究的产物，L4 的研究者发现，除了线程模型和线程间通信，内核的其他模块都可以移到用户空间^{[1][2]}。只要可以实现高效的进程间通信，就可以得到一个高效的微内核设计。于是他们提出了 Fast IPC，这是一种快速的进程间通信机制，基本思想是，尽量将所有消息都通过寄存器传递，它基本消除 Mach 性能的痼疾。只要针对每个系统实现一个高效的 FastIPC 机制，就可以为上层提供一个统一高效的接口。这个接口已经被定义为规范，并被几乎所有 L4 的实现采用^[5]。内存管理、文件系统和硬件驱动等都作为用户进程在用户空间实现，留下一个最小的内核。第二章详细微内核的发展。

按照 L4 理论思想实现的微内核，不仅获得了所有基于微内核设计的优势，而且性能出众。这些优势使得基于 L4 的微内核在不需要通用操作系统功能的嵌入式领域有巨大的发挥空间^[22]。随着物联网的兴起，人们对嵌入式系统的需求与日俱增。嵌入式系统复杂多变，且硬件性能各异，体积小、效率高的 L4 微内核可以很好的满足这些需求。本论文所设计的内核在 L4 理论之上增加了权能和容器，可以很好的保证系统的安全性，同时又为上层的软件设计提供了灵活性。这为嵌入式系统提供了一个高可靠、高性能、高灵活性、低成本的解决方案。

1.3 论文组织结构

大致介绍了论文关注的主要问题领域，给出了课题的目标和实验环境以及本论文的详细章节编排。

第1章主要介绍了微内核的概念以及 L4 的起源、核心概念、目前主流的实现及其优劣，然后探讨了 L4 的一些核心技术。

第2章分析了 Nros 的体系结构，通过本章可以对 Nros 的全貌有个整体的概念。

第3章详细介绍 Nros 各个模块的实现，详细介绍了线程模型、IPC、容器、内存等内容。

第4章详细介绍如何测试运行 Nros 内核，并在 Nros 之上开发简单应用程序。

第5章探讨了本论文的不足和展望。

第二章 微内核技术概述

2.1 微内核

微内核通过特殊的设计来最小化操作系统，它们通常只实现了操作系统所必需的非常小的一部分抽象和操作，比如特权管理、地址空间、线程和调度还有基于消息传递的进程间通信。所有原本在宏内核中必须出现的组件（比如驱动、文件系统、分页、网络等等）在微内核里都不会出现，这些组件全部运行于用户空间^[12]。系统组件实现成服务器的模式，应用程序使用 IPC 和共享内存的方式，通过一组定义完好的协议来通信。

在微内核基础上构建操作系统具有模块性、灵活性、可靠性，并且对于构建嵌入式系统，微内核还具有轻便的特点，微内核很小，这就使得移植微内核到新的平台要比移植 Linux 这样的通用操作系统要容易得多。然而，以 Mach 为代表的第一代微内核表现并不好，与其单内核的对等体性能落差太大，以致普遍对微内核方法产生了怀疑态度。为了提高性能，Mach 又把一些关键服务从用户空间重新放到内核空间，牺牲了微内核设计的益处。不过，微内核的研究并没有因此中断，经过仔细分析 Mach 性能低下的原因，研究人员发现，影响性能的并不是微内核的设计方式，而是最初的实现有问题。第一代微内核从单内核着手，想把单内核改造成微内核，而不是从头构建一个微内核。结果，第一代微内核的进程间通信的效率非常低下，大大超过了 CPU 的高速缓存所能容纳的大小，于是 CPU 频繁读取内存，导致性能严重下降。由此诞生了针对这些问题的第二代微内核，代表是 Exokernel^[7]和 Nemesis。

Exokernel 是由 MIT 于 1994 年到 1995 年研发的。Exokernel 的核心观点是，内核抽象限制了灵活性和性能，所以都应该被消除。Exokernel 的定位是一个安全的硬件层，为应用层提供基本操作原语，将抽象解放到用户空间，按其需要自己实现^[14]。L4 由德国科学家 Jochen Liedtke 于 1995 年开发，L4 的前身是 L3，没有 Exokernel 那么极端，但也非常强调性能。L4 通过最小化操作系统的核心功能来提供灵活性和性能。Nemesis 是由剑桥大学于 1993 到 1995 年开发的，旨在为 CPU、内存、磁盘和网络带宽等资源提供 QoS。

除了学术研究以外，从 1980 年开始，陆续有很多工业界开发并部署的微内核系统出现。两个比较有名气的是 QNX 和 GreenHills Integrity。QNX 是 1980 年左右做为 80x86 系列 CPU 开发的，后来又移植到了不同平台^{[11][41]}。GreenHills Integrity 是高度优化的商业嵌入式实时操作系统，该内核是抢占式的，并且中断

延时很低。与所有的微内核一样，QNX 和 Integrity 都是依赖用户空间的服务器来提供操作系统的功能（文件系统，驱动程序，网络栈），并且这些内核都非常小巧。

2.2 L4 微内核

L4 是第二代微内核，目标是高度灵活性和性能最大化，同时不牺牲安全性。L4 是目前微内核研究最活跃的分支。为了达到性能要求，L4 坚持从设计上来达到小的目的，小即快，L4 只提供了几个基本抽象机制控制整个内核：地址空间、内存映射、线程、调度、同步的 IPC。

由于强调小和灵活性，L4 在内核里使用 IPC 作为唯一的通信机制。基本的 IPC 机制不仅用于用户级线程传递消息，还用于传递中断请求、异步通知、内存映射、线程启动、线程抢占、异常和页错误。由于这种设计，尤其是其 IPC 的实现在性能上相比于第一代微内核的巨大改进，使得 L4 一经面世就受到广大关注，并不断优化完善直至今日。

2.2.1 L4 的起源

德国计算机科学家 Jochen Liedtke 从 1984 年开始研究 L3 微内核，他发现精心设计的系统和实现能够减少因为系统 IPC 开销过大而造成的性能损失，并开始论述如何设计高效的微内核系统^{[10][16][17]}。后来他在 IBM 的 Thomas J. Watson Research Center 和他的同事提出了 SawMill MultiServer 的 OS 架构，并发表论文详细描述了 SawMill Linux 的基本原理和实现方案。此后，想法逐渐发展成为今天广义的 L4 的架构。

2.2.2 L4 微内核规范

一个高性能的系统软件常常需要面对这样的问题：它在体系结构 A 表现优异，但是在体系结构 B 上却表现得不尽人意。任何系统软件的性能都与很多因素有关，因为性能本身就跟很多因素有关，设计只是其中的一部分。L4 也面临过这样的问题。第一次是从 80486 到奔腾系列，第二次是从 Intel 系列到 ARM 等非 x86 体系结构。

L4 解决方法是依靠微内核规范^[19]。该规范的设计需要满足两个明显冲突的目标。第一个是保证用户态软件的兼容性和可移植性；第二个是给内核工程师足够的余地优化性能、内存占用和耗电。该规范的手册里详细描述了：1) 硬件独立的 L4 API；2) 32/64 位 ABI；3) 用户空间可见的内核数据结构，如用户空间线程控制块（UTCB）和内核信息页（KIP）；4) 控制缓存和频率等 CPU 相关的功能；

5) IPC 协议; 6) 内存映射和用户控件中断处理等。

原则上, 每个 L4 微内核实现都需要遵守该协议, 但是实现上, 会衍生出很多版本。为了避免该种现象, L4 规范提供了一个测试套件, 用于验证的微内核实现与 L4 规范的吻合程度。

2.2.3 L4 的实现及其现状

L4 从最初的 L4/x86 演进到如今的 L4 家族, 已经从学术界成功跨越到了工业界, 大量部署在车载设备这样的工业设备上。上个世纪 90 年代末, 由于版权问题, L4 社区启动了 Fiasco 项目, 该项目在实现的过程中略微修改了 L4 规范的行为, 作为一个变体存在, 该实现使用了无锁和无等待的同步技术^[23]。DROPS (Dresden Real-time Operating System) 是运行于 Fiasco 之上的操作系统, 提供了实时性和内核抢占^[25]。Fiasco 从零开始实现, 这也使得它有更多的机会发现并解决一些 L4 实现过程遇到的问题, 例如: 时间片记法、优先级反转、优先级继承、内核抢占等。Fiasco 的解决方案并不完美, 增加了不少内核的复杂性和 IPC 开销。德累斯顿理工大学的操作系统研究小组 (TUDOS) 负责 Fiasco 的研发, 在目前所有还继续存在的 L4 项目中, Fiasco 是最早出现并获得关注的一个项目。此前, L4 都是用汇编语言实现的, Fiasco 是兼容 L4/X86 的 C++ 实现版本。此后 TUDOS 试图完全按照 SawMill 架构所描述的步骤来进行, 先开发 Fiasco 作为微内核, 然后开发了一系列的 Server 来完成一个真正的 OS 所要进行的工作, 包括任务管理, 内存分配, 时钟管理, 设备管理等等, 这些 Server 总称为 L4Env。他们已经完成了一个 OS 所具备的所有功能, 并开发了具有实时功能的窗口管理器 DoPE, 可以使用 DoPE 创建按钮, 对话框等各种桌面元素, 成为一个可用的 GUI。但是 L4Env 同真正的 Linux 相比, 易用性方面还是有很大的不足, 于是诞生了 L4Linux。L4Linux 将 Linux 移植到 L4 微内核上, 整个内核运行于用户态。从 L4Linux-2.0.x, 到 L4Linux-2.2.x, 再到 L4Linux-2.4.x, 到最新的 L4Linux-2.6.x 的时候, 更新速度已经接近正常的 Linux 发布的速度。这项工作带来的另一个好处就是实现了 Linux 的虚拟化, L4Env 的每一项功能都是以 Server 的形式提供, 因此可以同时服务给众多的 Client, 而不产生冲突, 自然就可以运行多个 Linux 了, 这为以后 TUDOS 在 Trusted Computing 方面的工作奠定了基础。目前 TUDOS 的工作重点包括: 实时系统, 系统安全以及虚拟化技术, 这三个方面都是基于微内核技术而来。参与 Fiasco 工作的团队远远没有 Pistachio 那么国际化, 基本上是 TUDOS 一家在作, 虽然从公开发表的论文来看, 也有一些大学在从事 Fiasco 相关的工作, 但是基本上都是在 L4Env 或者 L4Linux 层面, 很少有人能够切入 Fiasco 本身。Fiasco 遵从 GPL V2 协议, 目前 TUDOS 的 shepherd 是 Prof. Hermann Haretig 教授。TUD 目前

在 Trusted Computing 方面的工作，基于微内核进行 TC 架构的研发是 TUD 的微内核研究中的一个特色，目前相关研究都是从 L4/NIZZA 而来，L4/NIZZA 原本是为提高 VPN 的安全性而开发的一套架构，其基本原理就是利用同一台机器上的 2 个 L4Linux 来分割内网和外网，而网络协议栈从 L4Linux 中分离出来，直接运行与微内核之上，从而极大地提高了网络设备的安全性。Fiasco 也有类似于一般实时系统的资源预留机制（Resource Reservation），从而达到硬实时的效果。

与 Fiasco 同时开发的还有一个由 Liedtke 参与的内核项目 Hazelnut。这个微内核尝试使用面向对象语言 C++ 来实现操作系统。该内核也是一次成功，它证明了使用面向对象语言而不使用 C 或汇编来实现操作系统是可行的，并且性能是可接受的。

Pistachio 是第一个提供平台无关 API 接口的内核，Fiasco 以及之前的微内核实现都是平台相关的^[3]。尽管底层内核的概念一样，但是新的 API 对以前的 L4 做了很多修改，为多处理器提供了更好的支持，线程和地址空间松耦合，引入了用户控件线程控制块和虚寄存器。自 2001 年发布新版 L4 API 规范（Version X.2，或 V4）后，系统架构组（System Architecture Group）重新实现了一个新的微内核，即 Pistachio。该内核从头实现，以双重 BSD 协议开放。Pistachio 是目前开源的 L4 中支持平台最多的，目前可用的有 Alpha, AMD64, ARM, IA32, IA64, MIPS64, POWERPC32。

PikeOS 是一种基于 L4/x86 的商业微内核系统，面向军用、航空航天等领域等高安全性的应用^[21]。PikeOS 基于 ARINC653，ARINC653 是安全领域一个最基本的设计原则，其基本要求如下：1) Partition Management；2) Process Management；3) Time Management；4) Interpartition Communication；5) Intrapartition Communication；6) Health Monitoring。PikeOS 对于 L4 做了大量改进，因为最初的 L4 只是一个研究性质的内核，不是面向嵌入式设备的内核。第一个改进是在内存映射数据库（Memory Mapping Database），L4 采用了独特的递归内存管理模式，这个模式会导致映射数据库（Mapping Database）过大，查询或者更新映射数据库的时间不可预计，内存的 unmap 操作会导致复杂的映射数据库。一般来说，在高安全性内核中，动态的内存管理是不需要的，任何程序在启动的时候都已经规定了好使用的内存大小以及所使用的内存位置。因此，L4 的 Sawmill Method 对于高安全性内核的应用来说，就比较复杂。其实只要实现内存保护就足够了，但是 Sawmill Method 存在的意义在于它可以很容易地实现对 Linux 这类复杂系统的内存管理机制的模拟。PikeOS 改进了 Sawmill Method，使用了一个带头任务来管理内存，只有当所有子任务的内存全部释放之后，才 unmap 掉这些内存。

OKL4 是 Pistachio-Embedded 的延续，它目前由 Open Kernel Labs 公司维护，

但是研究工作基本上都是在 ERTOS 完成的。OKL4 的市场化很成功，已经有很多产品使用了 OKL4，包括基于 OKL4 的 OpenMoko 已经面世。

Coyotos 不是 L4，但是 Coyotos 和 L4 之间的关系之密切远远胜过了其他微内核和 L4 之间的关系。比如 Fast IPC, Capability-Based OS, IDL。Coyotos 是 KeyKOS 和 EROS(Extremely Reliable OS)的改进版本，从 EROS 的名称或许可以看出，这个系统和以上的系统有些不同，它强调 reliable，所以 EROS 刚开始的时候，被应用于一些军用系统，但是后来发现 Synchronous IPC 会导致一个 Denial of Service 的 Bug，这个 Bug 存在于所有基于同步 IPC 的系统中，当然也包括所有的 L4。Coyotos 的目标是提供具有军用级别的（EAL7 = Evaluation Assurance Level）的微内核，它使用一种新的称为 BitC（类似于 Haskell 语言的高安全性语言）来实现这个系统，并且整个系统采用面向对象风格开发。从概念上来讲，Coyotos 更为先进，权能也是在该系统上面首次被应用。

2.2.4 L4 的前沿

OKL4 Microvisor (OKL4 4.0)一款将 Microkernel 的高效精悍和 Hypervisor 的特性（提供一个真正的硬件虚拟层）相结合的商用操作系统^[53]。相对于 OKL4 Microkernel 3.0 来说，Microvisor 保持了其高效精悍实现的同时，为操作系统虚拟化提供了很为方便的实现，对于 Linux 内核的修改，比以前大幅度减少。Microvisor 4.0 方面，跟 Xen 在宣传上有些类似，比如 VCPU。

Microvisor 部分回归 Hypervisor，是不是意味着 Hypervisor 更优，该问题还值得讨论。Microkernel 其实是个普适的设计模式，任何软件系统的实现都可以按照 Microkernel 的思想去构建它的实现，而 Hypervisor 只是个为了实现虚拟化这个特定的功能而出现的一种方法。如果把 Hypervisor 按照 Microkernel 的思路去设计，其结果即是一个 Microvisor。

TU-Dresden 现在有两个 L4 Microkernel, Fiasco.OC 和 NOVA 都在持续进行，老版本的 L4/Fiasco 已经停止后续的维护和开发了，基于 L4/Fiasco 的 L4Linux 也停止更新很长时间。Fiasco.OC 最大的特点是实现了权能的完全支持，这个作为第三代微内核的一个核心技术已经在各个 L4 版本里面得到了彻底的执行，这个特点的支持其实不是内核的更新，而是完全重写。L4Env 也完全重写了，新的名称叫做 L4RE (L4 Runtime Environment)。Dice 也被另外一种类似与 C++ 的所完全代替。Fiasco.OC 的思路其实跟 OKL4 Microvisor 有些类似，就是在不要求特定的硬件支持的情况下（当然 MMU 还是必需的），实现更为简捷方便的 Linux 内核虚拟化的办法。

seL4 被称为第三代微内核。该内核研究人员已经使用 Haskell 语言形式化验证

了该内核。可以证明实现严格遵守高层设计，内核行为完全符合预期，永远不会崩溃，永远不会执行不安全的操作，并且可以精确预测内核的所有状态。seL4 针对实时应用，可潜在应用于强调安全和关键性任务的领域内，如军用和医疗行业。研究发现常用的攻击方法对 seL4 无效，如恶意程序经常采用的缓存溢出漏洞。Open Kernel Labs 首席科学家和 NICTA 的 ERTOS Group 负责人 Gernot Heiser 教授表示他们实现了很多广泛认为不可能实现的任务。

2.2.5 重要概念和技术

本节简要描述在论文后面会经常出现的几个概念，与通用操作系统相关的概念在此不再赘述，与 L4 微内核系统组件相关的部分会在描述分析到每一节时分别讲述。本将讲述共有的概念，不特别属于某一个组件。本节不探讨实现细节，论文后面几张会详细分析这些概念的实现。

1) 同步 IPC

IPC 处理线程间的信息交换，为了最大化性能，L4 IPC 采用同步方式：没有消息缓冲区，没有消息端口，内核内外没有二次拷贝。发送方通过系统调用发送一条消息，接收方要么立即接收到，要么让发送方等待。如果消息没有被发送，该消息也不会被置于缓冲区中，而是继续保留在发送方线程里。发送和接受使用的是线程标识符，或称线程 ID。IPC 可以传递其他参数，比如设置接收线程为任意线程等。

两个线程不能同时执行 IPC 调用，第一个调用者请求 IPC，线程阻塞，调度器从可运行队列中选取一个线程继续运行，刚才的调用者一直阻塞，直到对应的线程处理了信息完成通信。如果第一个调用者请求了一个非阻塞的调用，对方没有准备好，IPC 就会立刻退出，并返回错误。

IPC 主要有两种形式，用于不同环境。第一种成为标准 IPC，或称短 IPC (Short IPC)，该方式使用寄存器传递消息，速度非常快，但是可传递的数据量比较小。

L4 规范里规定了不同体系结构使用那些物理寄存器用作消息寄存器。第二种 IPC 是长 IPC (Long IPC)，这种方式使用 UTCB 内的数据缓冲区来存储消息。IPC 过程中，接收方从发送方拷贝该缓冲区。长 IPC 允许的消息长度很大。大段的数据传输可以使用多个 IPC，也可以使用共享内存。

2) 用户空间中断处理器

L4 将硬件中断封装成同步 IPC 消息，发送给注册到内核的中断处理器线程。这些中断处理器线程在系统启动的时候注册，每个中断一个线程。每个中断只会分到一个处理器线程，处理器线程可以一次处理多个中端，这是一种多对一的关系。时钟 TICK 中断是唯一一个运行在内核的中断处理线程。中断线程运行在

用户态，正在处理的中断会被屏蔽，但是其他中断是开启的，因此高优先级的中断可以抢占低优先级的中断。

3) Pager

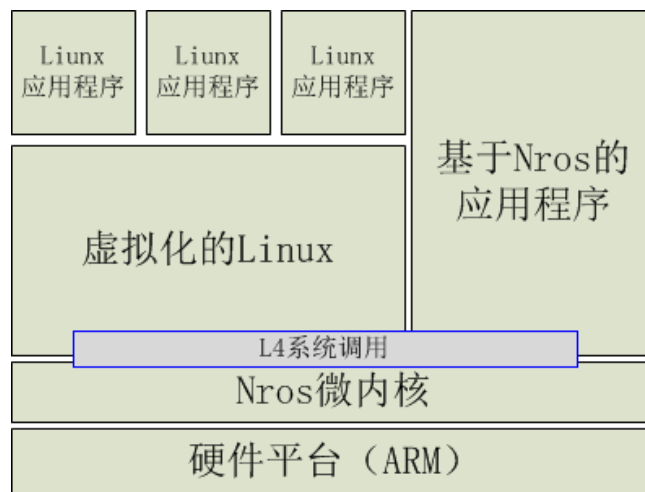
Pager 是 L4 特有的概念，它本身是一个线程，专职处理页错误消息和其他虚拟内存映射相关的操作。每个线程都有一个关联的 Pager，在此基础上用户可以自行构建复杂的模型。一个典型的模型是基于地址空间的管理器。Pager 是运行在用户态的。Pager 通常会实现成分层结构，最底层是物理内存，该层是特权级，其他层是用户级。所有的映射会被存储在一个映射数据库中，这种方式有时候会比较低效，已有许多不同的处理方式。以 L4 的观点来看映射数据库其实只是一个缓存，会在系统其他 Pager 之间传递分发，因此 Pager 反映了层级映射数据库。

第三章 总体设计

3.1 设计原则和特点

Nros 基于 L4 微内核理论，并使用了微内核理论的最新研究成果。Nros 遵循 L4 的基本原则，即将可以放到用户空间处理的任务全部移到用户空间处理，内核只提供三种基础机制：地址空间管理、线程和线程间的通信机制。

基于这个原则实现的 Nros 具有简单性、抽象性、高性能、高安全、高灵活性的特点。由于内核需要实现的机制只有三种，系统的复杂度非常低。整个 Nros 内核只有不到 10000 行 C 代码。基于 L4 的内核非常低层，三种机制都与体系结构密切相关，但 L4 对上层提供了相对一致的接口，用于上层构建复杂的系统，相当于针对不同体系结构提供了统一的硬件抽象层。这种抽象性，使得 L4 可以适用于实现各种有价值的应用，例如单一用途嵌入式系统、多任务实时操作系统、虚拟化平台，甚至在此基础上提供通用服务，实现一个通用的操作系统。Nros 的理想模型如图 3.1 所示。

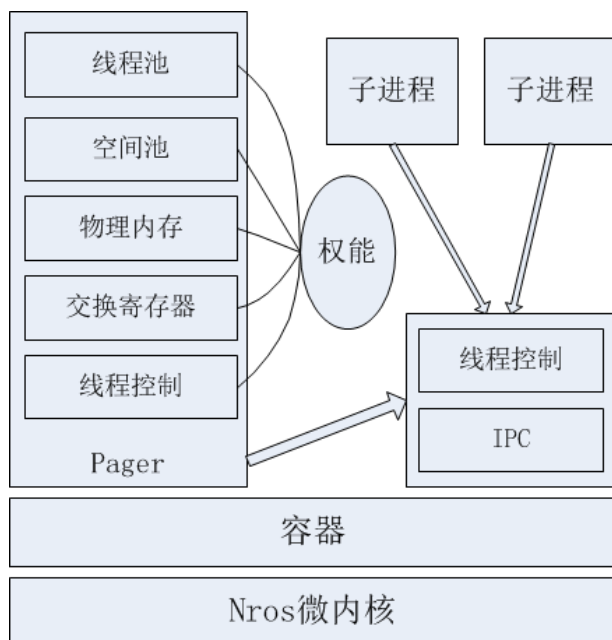


3.1 Nros 理想模型

在 Nros 之下是硬件平台，本论文的实现平台是 ARM，而 Nros 本身作为一个硬件抽象层存在。在此基础上，我们编写针对 Nros 的应用程序，即将 Nros 作为一个通用操作系统；也可以将修改客户操作系统（Guest OS），将 Nros 作为一个硬件平台，实现半虚拟化。

3.2 核心概念

本论文涉及一些核心概念，本节主要这些概念支撑着内核功能，其中一些概念是实际的组件，一些是功能。



3.2 Nros 核心组件交互

图 3.2 为 Nros 核心组件的交互图。Nros 区别与其他微内核架构的一个主要特征是容器。我们使用容器的概念来封装执行环境，是容器使得 Nros 同时具有虚拟化和构建通用操作系统的能力。容器关联一组权能，权能用于控制系统资源。Pager 是容器内用于管理内存（包括分配、映射和处理页错误等）的线程，同时主要由 Pager 来掌管容器的权能，每个线程会关联一个 Pager。线程之间包括 Pager 与线程之间都是通过 IPC 来通信。下面对各部分进行详细分析。

1. 线程

在 Nros 里，线程的概念与传统的操作系统并没有本质的区别，都是作为任务的执行环境，同时共享一些系统资源。这里将 Nros 的线程与 Linux 的线程做一个比较，以阐述 Nros 里线程的意义。

1) Linux 并不对进程和线程做区分，两者都是通过 task 这个概念来实现的，唯一的区别就是享有的资源不同。进程享有的资源多且有自己独立的地址空间，而线程会与其他线程共享地址空间可以文件描述符等资源。对于 Nros，线程更接近传统意义上线程的概念。线程会共享同一个地址空间，在 Nros 里不存在类比于 Linux 内核中 task 的概念，task 可以通过一组存在于相同地址空间的线程来实现。每个线程只存在于一个地址空间，每个地址空间可以包含多个线程。

2) 在 Linux 里, 线程分内核线程和用户空间线程两种。内核线程用于执行特权级的任务, 比如磁盘交换; 用户空间线程执行通用的应用程序任务。在 Nros 里, 不存在内核线程, 所有线程都运行在用户空间。线程间使用消息传递的方式通信, 执行系统任务 (比如文件操作) 和执行应用程序级的任务 (比如多线程执行某项计算任务) 没有任何区别。这里有一个例外, idle 线程是运行在内核空间的, 这是因为 idle 的机制与 CPU 非常相关, 出于效率的考虑将该线程放在内核空间运行。这是唯一一个运行与内核空间的线程。

2. 地址空间

地址空间用于界定一组线程的执行空间, 该空间内使用的是虚拟地址。每个地址空间都有一个独立的页目录, 一组权能, 还有一些锁相关的字段。同时, 地址空间也有自己的唯一标识, 内核维护一个链表, 可以通过唯一标志找到指定的地址空间。

地址空间主要支持四个操作: Map、Grant、Unmap、Flush。Map 操作用于与其他地址空间空间共享内存页。Grant 操作用于将指定的内存页移送给其他地址空间。Unmap 用于回收之前 Map 操作映射的内存页。Flush 操作用于页的所有者刷新缓存。

3. 容器

在 Nros 里, 容器提供了硬件体系结构相关的基础设施服务。容器用来隔离不同的执行环境, 每个容器包含一组地址空间、一个或多个线程, 还包括其他全局的系统资源, 例如虚拟内存和物理内存。

使用容器这种简单统一的概念, 使得在 Nros 之上可以使用任意的软件设计策略构建复杂系统。比如 client/server 模型或基于多线程的应用程序。

容器其实只是一种概念层次上的结构, 并不作为实体被硬件系统所认知。从 CPU 的角度来看, 只有线程和地址空间是真实存在的。

4. 权能 (Capabilities)

权能机制保护所有内核维护的资源。目前所有系统调用都被权能保护, 几个微不足道的系统调用除外。其他资源, 包括虚拟内存、物理内存、系统缓存、进程间通信及其他资源都受权能检查的保护。

权能机制是容器模型的基础设施, 是权能机制为 Nros 提供的安全性的保障。权能作用于容器的边界。系统还提供了权能控制的系统调用, 通过该系统调用, 权能可以被共享、授权和修改。这种机制最大化了权能的灵活性, 使系统的安全体系达到可配置的目的。

权能是一种单向的保护, 类似现实中的门的概念, 两个房间之间只有一扇门。作为发送者, 它必须拥有发送给接收者的权能, 接收者是目标资源。同时, 接收者自己也可以选择性的监听, 可以选择监听任意线程, 也可以选择监听某个线程

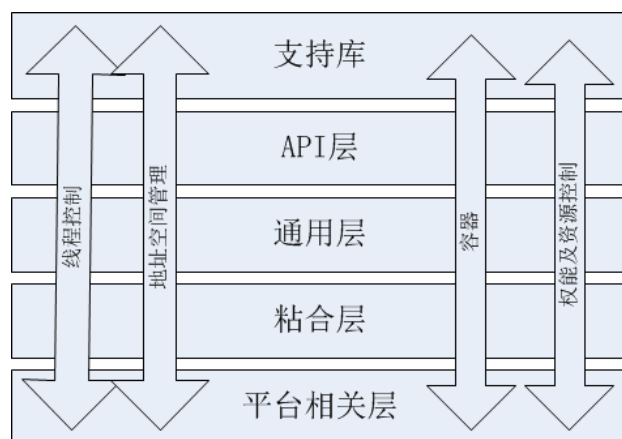
的消息。如果接收者选择监听的线程与发送者的线程不同，发送者就会一直阻塞。这是 Nros 设计的通信协议。

权能的检查从线程开始，如果线程不拥有该权能，将会向上检查地址空间，如果仍然没有，就会检查容器的权能。通过这种方式，几个线程可以共用一个地址空间的权能，不同的地址空间可以共享容器的权能。

权能的分类和实现方法请参看第四章。

3.3 Nros 的架构

L4 理论非常强调平台相关性，这是因为 L4 的诞生就是为了克服微内核架构的性能问题，为此在 L4 理论诞生之初即将针对不同硬件平台的优化囊括于理论之中。实现 L4 微内核的一个主要任务就是要编写大量平台相关的代码。包括系统的启动、虚拟内存管理、中断向量的设置、中断处理器的设置、异常的处理、IO 端口的初始化等工作。Nros 设计的一个目标，就是克服 L4 的微内核非常高的平台相关性。为此使用分层机制尽量减少平台相关的代码，尽可能多的复用 L4 接口相关逻辑的代码，使得将 Nros 移植到新的平台更加容易，同时保证内核的性能。此外，该内核使用容器的概念来实现不同的执行环境，使得 Nros 的能力多元化，既可以作为一个通用操作系统，又可以作为一个虚拟化平台，使用 L4 的系统调用修改 GuestOS，以实现半虚拟化。



3.3 Nros 的架构

图 3.3 为 Nros 的架构。该图也反映了 Nros 代码的组织结构。在实现上，每一层分别有一个文件夹，相同模块的功能在不同层次都有体现，但实现仅限于该层的功能。例如内存管理模块，在平台相关层主要提供与 CPU 相关的虚拟内存关闭、创建页表、设置页标志、刷新缓存等非常底层的服务；通用层使用粘合层实现了一套通用的虚拟内存映射结构，上层仅是根据需要提供更加丰富的封装接口。有些功能模块不会纵跨多个层次，比如初始化智慧设计平台相关层、粘合层。下

面几节分别阐述 Nros 各层的职能和交互关系。

3.3.1 平台相关层

平台相关层负责系统的初始化，为上层平台相关的服务。本论文实现的内核基于 ARM 平台，目前的实现版本支持的体系结构是 ARMv5。

初始化工作包括：a) 初始化内核栈；b) 设置控制寄存器状态；c) 全局页表，用于进入虚拟内存空间；d) 刷新 ttb，打开 mmu 的 cache 功能；e) 设置系统中断向量表。

平台相关服务包括：a) 异常处理机制；b) 系统调用机制，包括保护现场，跳回用户空间等操作；c) mutex 的实现；e) irq 的设置；f) 系统优化相关的库函数，例如内存拷贝；g) 虚拟内存映射；h) 内存管理单元 mmu 相关操作。

除此之外，一些编译链接的参数也会在这一层设置。

3.3.2 通用服务层

通用服务层实现了 Nros 内核所使用到的最核心的组件，包括：

1) 容器。这部分实现了容器的创建、初始化、删除、在容器内查找线程等操作。这部分还实现了 Pager。在 Nros 里，Pager 本质就是一个 UNIX 平台通用的 ELF 文件格式的线程。所有的 Pager 都是静态编译链接的，Pager 根据初始化信息设置自己的状态，并由容器启动运行。Pager 负责管理容器内其他线程的地址空间，包括页错误，此外 Pager 也管理自己的一组权能，可以被子线程共享。一个容器内可以包含多个 Pager，在容器初始化阶段根据静态信息顺序加载并启动。是容器的实现在 4.2 节详细讲述。

2) 权能。Nros 所有重要资源和重要操作都由权能保护，例如地址空间和 map 操作。这部门实现了所有系统调用的权能检查、按权能类型查看权能列表等操作。权能的实现在 4.3 节详细讲述。

3) 地址空间。这部分实现了地址空间的创建、删除、添加、查找等操作。地址空间的时间在 4.4 节详细讲述。

4) 线程。这部分定义了线程的结构，实现了线程创建、删除等操作。线程的实现在 4.5 节详细讲述。

5) 资源管理。系统所有的资源都由这部分统一管理，包括所有的权能列表、内存池、互斥锁、id 池、全局页表等。

6) 调度。这部分实现了与 Linux 相似的 runqueue 结构和调度算法。调度算法下一章会有详细介绍。

7) IRQ。实现了通用的 IRQ 接口。

除此之外，通用层还实现了启动期内存管理、时间和内核抢占以及一些调试方法。

3.3.3 粘合层

粘合层介于平台相关层和通用层之间，用于粘合不同体系结构与通用层之间的差异，同时又将最贴近机器的相关操作隔离在平台相关层。

粘合层封装了平台相关的系统调用方式、和内存管理接口。由于粘合层的既了解底层硬件平台，又了解高层所需资源，所以粘合层成了初始化系统的平台无关部分的最佳切入点。

3.3.4 API 层

API 层实现了 L4 规范的接口，并做了一些扩展。Nros 实现的接口如下：

1) int l4_ipc(l4id_t to, l4id_t from, unsigned int flags)

线程间通信，从 from 发送消息到 to

2) int l4_thread_switch(void)

线程转换，主动放弃 CPU

3) int l4_thread_control(unsigned int flags, struct task_ids *ids)

修改线程状态和信息

4) int l4_exchange_registers(struct exregs_data *exregs, l4id_t tid)

读取或修改挂起状态线程的上下文

5) int l4_unmap(unsigned long virtual, unsigned long npages, unsigned int tid)

取消映射某区段虚拟内存

6) int l4_irq_control(unsigned int req, unsigned int flags, l4id_t id)

线程可以使用该接口注册/取消设备 irq，可以选择是异步还是同步操作。

7) int sys_map(unsigned long phys, unsigned long virt, unsigned long npages, unsigned int flags, l4id_t tid)

映射虚拟内存

8) int l4_getid(struct task_ids *ids)

得到线程的线程 id

9) int l4_capability_control(unsigned int req, unsigned int flags, void *addr)

权能的控制

10) int sys_time(struct timeval *tv, int set);

查询系统时间，格式与 UNIX 系统相同

3.3.5 支持库

尽管 L4 理论提倡简单的接口，但它并不提倡直接使用这些库做应用程序的开发；相反，通常的做法是在这套接口之上构建一套软件库，实现诸如 POSIX 这类平台级的通用库来方便应用程序快速开发。

实现一套完整的 POSIX 接口是一项浩大的工程，其代码量远远超过 Nros 自身的代码。固本次实现并没有实现所有 POSIX，仅是实验性的实现了几个最常用接口，诸如 `fork()`, `exit()`, `execve()`, `mmap()`, `time()`。尽管仅仅实现这几个接口，直接使用最底层的 L4 接口也非常不方便，为此 Nros 在已实现的 10 个接口之上又实现和封装了一层名为 `libl4` 的用户层接口，一套针对设备驱动的名为 `l4dev` 的接口，一套针对内存管理的 `libmem` 接口，还一套移植过来的 `libc` 库。

下面分别介绍一下这几个库：

1) libl4

该库主要为底层 l4 接口纷繁复杂的参数提供功能单一的简化接口，例如：

```
int l4_irq_wait(int slot, int irqnum);
```

该接口主要封装了底层接口：

```
int l4_irq_control(unsigned int req, unsigned int flags, l4id_t id)
```

这种封装使得添加 irq 处理器非常清晰。

2) libdev

目前的 Nros 在设备驱动这一层的功能还比较简单，仅能处理 UART、定时器、键盘、LCD 这四种简单的设备。

a) UART 的接口：

```
void uart_tx_char(unsigned long uart_base, char c);
```

```
char uart_rx_char(unsigned long uart_base);
```

```
void uart_set_baudrate(unsigned long uart_base, unsigned int val);
```

```
void uart_init(unsigned long base);
```

b) 定时器的接口：

```
void timer_start(unsigned long timer_base);
```

```
void timer_load(u32 val, unsigned long timer_base);
```

```
u32 timer_read(unsigned long timer_base);
```

```
void timer_stop(unsigned long timer_base);
```

```
void timer_init_one-shot(unsigned long timer_base);
```

```
void timer_init_periodic(unsigned long timer_base, u32 load_value);
```

```
void timer_init(unsigned long timer_base, u32 load_value)
```

c) 键盘的接口:

```
void kmi_rx_irq_enable(unsigned long base);
```

```
int kmi_data_read(unsigned long base);
```

```
char kmi_keyboard_read(unsigned long base, struct keyboard_state *state);
```

```
void kmi_keyboard_init(unsigned long base, unsigned int div);
```

3) libmem

该库实现了与 Linux 核心相似的 `kmalloc()`, `kfree()`, `page_alloc()`等接口, 用于用户态的系统级操作。

第四章 实现

本章详细介绍 Nros 各模块的实现细节, 首先从全局来描述模块的功能和概念, 然后结合当前主流的实现方式, 说明选择某种实现方式的原因和意图。对于重要的模块, 还会解释模块使用的数据结构和接口。每个模块都会深入到代码, 阐述核心算法。

4.1 线程和调度

4.1.1 概述

线程是微内核支持的执行环境实体, 内核为每个线程分配一个全局的线程 ID, 每个线程会关联一个地址空间和一个页错误处理线程 **Pager**。此外, 用户空间线程控制块也映射在线程地址空间内。

线程在创建之初可以有两种初始状态, 分别是活动状态 (**active**) 和非活动状态 (**inactive**)。线程若要处于活动状态, 必须关联一个 **Pager**; 非活动状态的线程的创建有两个目的: 1) 创建之后修改线程的地址空间; 2) 为新线程分配已有地址空间。非活动状态的线程可以使用 `l4_exchange_registers()` 系统调用来修改其状态。

Nros 使用了一个非常简单的基于优先级的调度器来调度线程^[37]。其本身与 Linux 的 $O(1)$ 调度器很相似, 但是要简单的多。

4.1.2 线程的结构

线程实现于内核, 它的结构体为 **ktcb**, 表示“kernel task control block”, 与 Linux 类似, Nros 不明确区分任务 (**task**) 和线程的 (**thread**) 的含义, 下面为了与实现一致, 下文出现的“任务”皆意指“线程”。

ktcb 结构体主要字段如下:

```
struct ktcb {
    task_context_t context;           // 用户上下文
    syscall_context_t *syscall_regs; // 系统调用上下文

    struct link rq_list;              // 等待队列
    struct runqueue *rq;
```

```
l4id_t tid;                // 线程 ID
l4id_t tgid;               // 线程组 ID

int affinity;              // CPUaffinity
unsigned int flags;        // 任务状态标志
unsigned int ipc_flags;    // IPC 标志

struct mutex thread_control_lock; // 控制锁
struct spinlock thread_lock;      // 自旋锁

u32 ts_need_resched;        // 标志
enum task_state state;      // 任务状态

struct link task_list;      // 任务列表
unsigned long utcb_address; // 指向用户空间 UTCB 地址

u32 kernel_time;           // 调度相关字段
u32 user_time;
u32 ticks_left;
u32 ticks_assigned;
u32 sched_granule;
int priority;

int nlocks;

unsigned int exit_code;     // 线程退出 code

struct address_space *space; // 线程所在地址空间
struct container *container; // 线程所在容器
struct ktcb *Pager;        // 线程所关联的 Pager
int nchild;

struct waitqueue_head wqh_recv; // 消息发送接收等待队列
struct waitqueue_head wqh_send;
```

```

l4id_t expected_sender;

struct waitqueue_head wqh_notify;    // IRQ 通知等待队列
struct waitqueue_head wqh_Pager;    // Pager 等待队列

struct spinlock waitlock;            // 等待队列自旋锁
struct waitqueue_head *waiting_on;
struct waitqueue *wq;

unsigned long extended_ipc_size;
char extended_ipc_buffer[];
};

```

该结构的字段与教科书中所描述的任务控制块很相似，但是有几个不同的地方：a) 增加一个 **UTCB** 字段用于 **IPC** 时期传递消息数据；b) 为发送消息和接收消息分别设置了等待队列；c) 关联 **Pager** 线程，而不再由内核为线程分配内存和处理页错误 (**page fault**)；d) 增加一个 **container** 字段，关联任务所在的容器。

全局线程 **ID** 的概念是 **L4** 理论特有的。每个线程会分配到一个唯一的全局 **ID**。一个线程组只能属于一个单一的线程，称为 **leader**；一个线程只能同时隶属于一个线程组。每个全局 **ID** 可以用来定义一个唯一的线程组 **ID**。线程本地 **ID** 用于在线程组的 **UTCB** 区域中查找线程的 **UTCB** 结构。

4.1.3 线程的生命周期

在 **Pager** 初始化之初，线程都由 **Pager** 来创建，并为线程分配一个独立的地址空间，线程和其关联 **Pager** 通过 **IPC** 来通信。线程可以继续创建其他线程，通过通过 **IPC** 收发消息，这个过程中线程可能处于等待、挂起等状态。当一个线程需要退出时，它会发送退出消息给关联的 **Pager**，由 **Pager** 来负责销毁线程。

下面分别描述各环节的实现和执行过程。

1) 线程创建 (**thread_create**)

线程创建会以父线程的线程 **ID** 和地址空间 **ID** 为参数，传递到线程创建函数。得到相关 **ID** 之后，系统依次执行如下步骤完成线程的创建：

- a) 检查地址空间标志，判断是共享、创建还是拷贝父线程地址空间
- b) 分配一个新的 **ktcb** 结构

c) 根据标志设置线程的地址空间：如果是共享地址空间，则关联父线程地址空间到子线程；如果是拷贝，则使用拷贝方式创建一个新的地址空间，并关联；

如果是创建，则创建一个新的空地址空间，并关联

d) 设置 Pager 和 container

c) 设置新创建线程的线程 ID 和地址空间 ID

d) 体系结构相关的初始化：如果是新创建的则返回；如果是共享或拷贝，则需要复制系统调用寄存器上下文，并处理调度相关的时间片信息

e) 将新创建的 ktcb 加入到线程列表里

2) 线程运行 (thread_run)

线程运行仅仅是将线程移到运行队列里，由调度器调度运行。下面按调用栈简单说明这个过程：

a) 调用 sched_resume_async(task);

b) 设置线程状态为可运行：task->state = TASK_RUNNABLE;

c) 将任务加入到运行队列：sched_rq_add_task(task);

3) 线程挂起 (thread_suspend)

在 Nros 里，线程挂起是使用信号实现的，挂起的线程进入 INACTIVE 状态。

实现很简单：

```
int thread_suspend(struct ktcb *task) {
    return thread_signal(task, TASK_SUSPENDING, TASK_INACTIVE);
}
```

4) 线程销毁 (thread_destory)

线程的销毁必须在 Pager 里调用，如果需要销毁的是子线程，则调用 thread_destory_child(task) 销毁。该函数会递归销毁其子线程。当无子进程时会销毁自身，在销毁之前，会先唤醒等待该线程的其他线程。

5) 线程回收 (thread_recycle)

回收 ktcb 相关的资源。过程如下：

a) 从全局线程列表里移除该 ktcb：tcb_remove(tcb);

b) 将上下文信息清空，重新初始化到初始状态：

```
memset(&tcb->context, 0, sizeof(tcb->context));
```

c) 清空页表：

```
remove_mapping_pgd_all_user(tcb->space, &current->space->cap_list);
```

d) 再将该 ktcb 插入全局线程列表以供复用：tcb_add(tcb);

6) 线程等待 (thread_wait)

等待某个线程结束，然后唤醒等待在该线程上的其他线程，最后将线程放置到僵死线程队列里。

4.1.4 线程上下文转换

上下文转换跟体系结构密切相关，并且其效率对于系统的性能至关重要。Nros 的上下文切换使用会用汇编语言实现。Nros 基于 ARM 平台，理解这部分的分析需要具备 ARM 平台的知识。

用户态在调用系统调用或中断请求到来时会切换进内核态，内核处理相关请求后会进行一次调度，从运行队列里选择优先级最高的线程切换到该线程的用户态。进入内核态只需进行一些保护现场的工作。下面主要讲解系统如何由内核态切换到用户态。

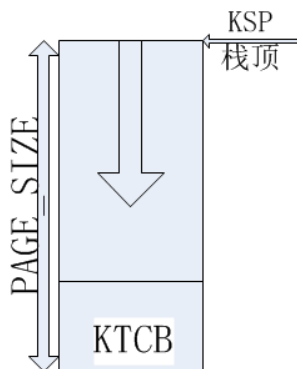
跳转到用户态的核心代码为：

```
void switch_to_user(struct ktc *task) {
    arm_clean_invalidate_cache();
    arm_invalidate_tlb();
    arm_set_ttb(virt_to_phys(TASK_PGID(task)));
    arm_invalidate_tlb();
    jump(task);
}
```

前面几个函数用于处理 cache，最后调用 jump() 返回用户态。下面详细说明 jump() 的实现方式：

```
void jump(struct ktc *task)
{
    __asm__ __volatile__ (
        "mov    lr, %0\n"          /* i1.将上下文放到寄存器 lr 里 */
        "ldr r0, [lr]\n"          /* i2.读取 spsr 到 r0 */
        "msr    spsr, r0\n"       /* i3.设置 spsr 为 ARM_MODE_USR */
        "add    sp, lr, #1\n"      /* i4.重设 SVC 栈 */
        "sub    sp, sp, #2\n"      /* i5.对其栈边界 */
        "ldmib  lr, {r0-r14}^\n"   /* i6.加载所有用户寄存器 */
        "nop    \n"               /* i7.等一个周期 */
        "add    lr, lr, #64\n"     /* i8.设置 PC */
        "ldr lr, [lr]\n"          /* i9.加载 PC 到 lr 寄存器 */
        "movs   pc, lr\n"         /* i10.设置 PC，跳转到用户态 */
        :
        : "r" (task), "r" (PAGE_SIZE), "r" (STACK_ALIGNMENT)
    ); }
```

该函数使用内联汇编实现，便于同 C 语言共同维护。首先需要说明一下内核栈的使用方式：在 Nros 里，每个线程有一个内核栈（kernel stack），用于在内核态运行。系统目前使用的内核栈大小是一个内存页。ktcb 结构也存储在该页中，该内存也的结构如下：



4-1 内核栈结构

即 ktcb 处于内核栈页的低端，而内核栈从顶端向下生长，换言之：

内核栈的大小 = `PAGE_SIZE - sizeof(struct ktcb)`

这种方式的实现使用了 C 语言的一个技巧，Nros 使用如下结构定义内核栈：

```
union ktcb_union {
    struct ktcb ktcb;
    char kstack[PAGE_SIZE];
};
```

下面详细分析上面切入内核态的过程（按注视中 ix 标号逐条解释）：

i1) 将 ktcb 指针放入 lr 寄存器中

此处%0 会有编译器替换为指针 task，即该线程的 ktcb 指针，ktcb 结构体的第一个字段是 task_context_t，该结构体定义如下：

```
typedef struct arm_context {
    u32 spsr;    /* 0x0 */
    u32 r0;      /* 0x4 */
    u32 r1;      /* 0x8 */
    u32 r2;      /* 0xC */
    u32 r3;      /* 0x10 */
    u32 r4;      /* 0x14 */
    u32 r5;      /* 0x18 */
    u32 r6;      /* 0x1C */
    u32 r7;      /* 0x20 */
    u32 r8;      /* 0x24 */
};
```

```

u32 r9;      /* 0x28 */
u32 r10;     /* 0x2C */
u32 r11;     /* 0x30 */
u32 r12;     /* 0x34 */
u32 sp;      /* 0x38 */
u32 lr;      /* 0x3C */
u32 pc;      /* 0x40 */
} __attribute__((__packed__)) task_context_t;

```

该结构体包含了用户进入内核态需要保存的所有信息，

i2) 将 `spsr` 存入 `r0`

从 i1 描述的结构可以知道，`ktcb` 的最低 4 个字节保存的是 `spsr` 寄存器。

i3) 恢复 `spsr` 寄存器标志到用户态

i4) 恢复内核栈到内存页最高位置，即重置内核栈

```
spsr = lr + PAGE_SIZE
```

即：

```
spsr = *ktcb + PAGE_SIZE
```

i5) 恢复用户态寄存器

i7) 设置 `lr` 指向用户态 `pc` 寄存器位置，十进制数 64 即寄存器 `pc` 在 `task_context` 里的偏移

i8) 加载 `pc`

i9) 跳转回用户态

4.1.5 线程调度

1) 调度器相关数据结构

```

struct runqueue {
    struct scheduler *sched;
    struct spinlock lock;      /* 自旋锁 */
    struct link task_list;     /* 任务列表 */
    unsigned int total;        /* 总线程数 */
};

struct scheduler {
    unsigned int flags;
    unsigned int task_select_counter;
    struct runqueue sched_rq[SCHED_RQ_TOTAL];

```

```

struct runqueue *rq_runnable;    /* 可运行任务队列 */
struct runqueue *rq_expired; /* 时间片已过期的可运行任务 */

struct ktcb *idle_task;
int prio_total;                  /* 优先级总和 */
};
预定义的任务的优先级:
#define TASK_PRIO_MAX            10
#define TASK_PRIO_REALTIME      10
#define TASK_PRIO_PAGER         8
#define TASK_PRIO_SERVER        6
#define TASK_PRIO_NORMAL        4
#define TASK_PRIO_LOW           2
#define TASK_PRIO_TOTAL         30

```

3) 调度算法概述

Nros 的调度器跟 Linux 的 O(1)调度器很相似。线程的优先级决定了线程可用的时间片。系统会计算每个线程的优先级占所有线程的优先级总和的一个比率。当总和变化的时候（例如，线程死亡或者创建）就会重新为每个线程计算一次时间片，此时不论线程的状态，系统假设所有线程都是可运行状态。一旦所有可运行线程的时间片都过期了，runqueue 就会交换（rq_runnable 和 rq_expired），处于 SLEEPING 状态的线程会从 rq_runnable 队列移除并放到队列的后面，时间片不变。挂起的线程需要重新计算时间片，因为可以认为这些线程已经处于 INACTIVE 状态很长时间了^{[43][47]}。

4) 调度算法实现

实现该算法主要有三个函数，分别为：

a) sched_select_next()

核心逻辑为：

```

if (sched->flags & SCHED_RUN_IDLE) {
    // 判断是否是需 IDLE
    next = sched->idle_task;
    break;
} else if (sched->rq_runnable->total > 0) {
    // 如果 runnable 队列不空，则选择该队列头部的任务
    next = link_to_struct(sched->rq_runnable->task_list.next,

```

```

        struct ktcb, rq_list);

        break;
    } else if (sched->rq_expired->total > 0) {
        // 如果为空，首先交换时间片过期队列和可运行队列，再选择
        sched_rq_swap_queues();
        next = link_to_struct(sched->rq_runnable->task_list.next,
                               struct ktcb, rq_list);

        break;
    } else if (in_process_context()) {
        // 如果两个队列都为空，则直接运行 IDLE 任务
        next = sched->idle_task;
        break;
    } else {
        // 如果当前线程不是运行任务的线程，则一定是执行 IRQ 的任务，
        需让本次 IRQ 处理继续执行完
        next = current;
        break;
    }
}

```

b) sched_prepare_next()

该函数首先会判断是否是第一运行，首次运行的线程的时间片为 0。如果是首次运行，则使用 sched_recalc_ticks() 来计算它的时间片。Nros 不会每次为所有线程一次性的计算时间片，它会用此种方式，每次线程初次运行的时候计算。

c) sched_recalc_ticks()

该函数计算实际的时间片，算法非常简单，简单到实现可以时使用 inline 来优化。核心代码只有一行：

```
task->ticks_assigned = SCHED_TICKS * task->priority / prio_total;
```

这里 SCHED_TICKS 为时钟每秒产生的 tick 数，默认为 1000。这也即是说，最多 1 秒钟，所有线程都会被调用一次。此时 rq_runnable 队列上所有线程的时间片都会用光，rq_runnable 和 rq_expired 会进行一次交换，所有线程的时间片也会重新进行一次计算。

4.2 线程间通信（IPC）

4.2.1 概述

IPC 是 L4 理论的核心，而消息传递是 IPC 的核心。IPC 是微内核提供的为数不多的系统调用之一，任何其它功能都必须构建于线程之上，实现为线程服务器，这些服务器与它的客户端使用消息传递进行通信。服务器和客户端仅是一个相对的概念，通常接收者是服务器，发送者是客户端。

在 L4 理论中，所有的 IPC 调用都是同步的，并且无缓存。同步的 IPC 就需要发送者和接收者之间通过某种形式的参数来传递信息^{[31][38]}。一种显而易见的参数传递方式即一段可以被发送者和接收者同时使用的缓冲区。如果发送者或接收者尚未处于可响应状态，另一方就必须等待，此时系统会进行调度，让其他线程继续运行。无缓存的 IPC 减少了拷贝所需的时间，这使得 L4 微内核可以进行高效 IPC 来满足性能要求^{[44][46]}。

4.2.2 消息类型

Nros 实现了两种消息传递类型，名为 short_copy 和 full_copy，分别对应短 IPC（short ipc）和全 IPC（full ipc）。详细说明 Nros 的 IPC 实现方式之前，需要先讨论两个问题：线程本地存储（TLS）和用户线程控制块（UTCB）。

与任何 L4 微内核一样，Nros 也需要一种访问用户线程控制块的方式，在 IPC 过程中传递参数。有几种典型的可行方法：

1. 在每次上下文转换的时候，为每个线程的虚拟地址空间映射一段私有的物理 TLS 页
2. 维护一个指向该区域的全局指针。调度器在每次上下文转换的时候更新该指针，使其指向当前的 TLS。
3. 把这些都放到用户空间，让运行时库来做这些事情。这样做的话，需要做如下事情：
 - 在地址空间内预定义一个固定的 TLS 表
 - 为该空间内的线程分配唯一 ID
 - 得到当前线程 ID
 - 加锁
 - 根据 ID 查找到与该线程关联的 TLS 块
 - 释放锁
 - 返回该块的地址

第三种方案是最缺乏灵活性的方案，因为这需要管理 TLS 区域，还有并发问题，TLS 的数量也有限制。下面具体分析一下前两个方案：

第一种方案：这种方式对于用户空间是最方便的，因为应用程序可以直接引用指向 TLS 的指针。这种方式不需要为每个线程分配一个独立的页来存储线程本息数据。

第二种方案：在上下文切换的时候，映射 per-thread 物理页到一个固定的虚拟内存页是最灵活的，因为 Pager 在创建新线程的时候顺便分配一个物理页很容易。在运行时后做映射。缺点是为每个线程浪费一个页。

Nros 采用的方案是混合这两种方法。在上下文转换时，Pager 会映射线程私有的 TLS 页，同时用一个叫做 UTCB 的结构体指针指向那个页，如此，线程就可以直接使用 TLS 的数据，并且不需要为每个线程设置一个固定的地址。页也可以复用，如果一个 UTCB 只有内存大小的四分之一，那么一个页就容纳四个 UTCB。

UTCB 是一个 per-thread 数据结构，用于为 IPC 消息寄存器和私有数据存储线程本地信息。UTCB 位于线程地址空间内，是一个虚拟地址，该地址在每个线程中是唯一的。可以在运行时通过读取 KIP（内核接口页）的 utcb 字段来找到它的地址。

UTCB 存储消息寄存器，该寄存器在 IPC 的时候在发送者和接收者之前传递。视此次 IPC 是发送还是接受，消息寄存器域会有不同的使用方式。如果是发送，消息寄存器用于传递参数；如果是接收，消息寄存器会被其它线程写入参数。UTCB 还可以用于存储只有线程自身可见的私有数据。

UTCB 的结构体如下：

```
struct utcb {  
    u32 mr[MR_TOTAL];  
    u8  notify[TASK_NOTIFY_SLOTS];  
    u32 mr_rest[MR_REST];  
};
```

其中 mr[MR_TOTAL]是消息寄存器。在 ARM 体系结构有六个这样的寄存器，这些寄存器在 IPC 的过程中会映射到实际的 ARM 寄存器。notify[TASK_NOTIFY_SLOTS]用于线程注册 IRQ 通知槽。mr_rest[MR_REST]存储消息寄存器之外的数据，只在全 IPC 时候才会使用。

UTCB 的地址和内存分配都在用户空间处理，微内核不做干涉。管理 UTCB 地址有两种场景。第一种，如果 Pager 需要管理多个有独立地址空间的子任务，它需要自己实现一组 UTCB 管理函数来管理位于不同地址空间的 UTCB。这也意味着 Pager 需要在分配 UTCB 的时候区别对待不同的地址空间。例如 UTCB 空间可能小于内存页的大小，位于相同地址空间的另外一个线程可能会将自己的

UTCB 也放入该页，但其他地址空间可能处于安全的考虑需要将 UTCB 放如单独的内存页，例如设置不同的页标志。另一种清醒是，Pager 只是一个简单的多线程应用程序，它的所有子线程都共用 Pager 的线程。这种情况下，由于 UTCB 位于共享的地址空间，Pager 可以使用某种内存池的方式来自己管理 per-thread 的 UTCB 区域。

UTCB 可以包含在任何虚拟地址空间内，但是有一个限制：它们不能重叠。因为在 IPC 的过程中，内核会直接拷贝不同 UTCB 内的数据，如果他们的地址重叠了，数据就会被破坏。换言之，进行 IPC 的两个线程，必须保证它们的 UTCB 区域不会重叠。

4.2.3 消息拷贝方法和效率

上一节提到，Nros 使用两种消息传递方法，分别为短 IPC 和全 IPC，对应这两种 IPC 有两个消息拷贝的方法，分别为短拷贝和全拷贝。

1. 短拷贝（short copy）伪代码如下：

```
ipc_short_copy(ktc* to, ktc* from) {
    mrs_src = KTCB_REF_MRS(from);
    mrs_dst = KTCB_REF_MRS(to);

    memcpy(mrs_dst, mrs_src, MR_TOTAL);
    return;
}
```

宏 KTCB_REF_MRS 从 ktc 结构体找到 utcb 结构，然后从 utcb 中找到消息寄存器的起始地址，最后将所有消息寄存器从 from 拷贝到 to 的 utcb 寄存器域中。

2. 全拷贝（full copy）伪代码如下：

```
ipc_full_copy(ktb* to, ktb* from) {
    utcb *from_utcb = (cast to utcb *)from->utcb_address;
    utcb *to_utcb = (cast to utcb *)to->utcb_address;
    int ret;

    // 首先做一次短拷贝
    if ((ret = ipc_short_copy(to, from)) < 0)
        return ret;

    // 做全拷贝
```

```

memcpy(to_utcb->mr_rest, from_utcb->mr_rest,
       MR_REST * sizeof(unsigned int));
}

```

全拷贝仅仅是将 `utcb` 结构体中的 `mr_rest` 也一并拷贝到目标 `utcb` 中。尽管这两种拷贝方法都很简单并且很相似，但它们在性能上的差别却很大。短拷贝可以完全利用 CPU 的寄存器来复制，而全拷贝则必须要读写内存，所以不适合频繁调度的场景。利用寄存器的短拷贝是 L4 理论强调的关键技术，下面详细描述在 ARM 平台上，Nros 是如何利用寄存器实现高效数据传递的。

首先介绍 ARM 是如何分配给 IPC：

a) `r0 - r2`: 作为传递给 `ipc()` 调用的参数。这三个寄存器由微内核读取，它们具有系统级的意义。系统级意味在这里代码是遵守 APCS 的。APCS 的意思是 ARM 过程调用标准 (ARM Procedure call standard)。该标准规定，只有 `r0-r3` 可以用于参数传递，超过该数目的参数需要放置在栈上或通过指针访问。

b) `r3 - r8`: 这六个寄存器会映射到 `utcb` 里定义的 `MR0-MR5` 六个主消息寄存器。它们的格式内核不做要求，由应用程序自行设置。有一个例外，如果 IPC 的接收者设置接收线程为 `ANYTHREAD`，内核会设置预定义的 `MS_SENDER` 为其发送者，传递消息并不会映射到寄存器，这样做是由于复杂度比较高。

c) `l4lib` 寄存器: (`MR_TAG`, `MR_SENDER`, `MR_RETURN`)

辅助库 `l4lib` 会使用一些消息寄存器来进行大部分 IPC 操作。例如，每个 IPC 会设置一个 `tag` 标记此次 IPC 的原因。同样 `send/receive` 操作需要返回值。发送给所有线程的消息，接收者那一边需要知道发送者的 ID；接受所有线程消息的线程也需要知道消息的来源。这些都有系统预定义的寄存器来传递，但内核事实上并不需要知道这些，这些都是在库一级实现。

d) 系统调用寄存器: `L4SYS_ARG0` 到 `L4SYS_ARG4`。这些寄存器用于传递系统调用的参数。`L4SYS_ARGx` 的定义如下：

```

#define L4SYS_ARG0  (MR_UNUSED_START)
#define L4SYS_ARG1  (MR_UNUSED_START + 1)
#define L4SYS_ARG2  (MR_UNUSED_START + 2)
#define L4SYS_ARG3  (MR_UNUSED_START + 3)

```

这里宏 `MR_UNUSED_START` 指向的是系统调用寄存器上下文中未使用的寄存器的偏移量，系统调用上下文结构体定义如下：

```

typedef struct syscall_context {
    u32 spsr;
    u32 r0;
    u32 r1;
}

```

```

u32 r2;
u32 r3;    /* MR0 */
u32 r4;    /* MR1 */
u32 r5;    /* MR2 */
u32 r6;    /* MR3 */
u32 r7;    /* MR4 */
u32 r8;    /* MR5 */
u32 r9;
u32 r10;
u32 r11;
u32 r12;
u32 sp_usr;
u32 lr_usr;
}__attribute__((__packed__)) syscall_context_t;

```

宏 `MR_UNUSED_START` 即指向 `r0`，从这里可以看到 Nros 的系统调用是如何使用寄存器的。在实现 POSIX 系统调用时，即使用这种方式。

4.3.4 IPC 实现

IPC 作用于两个线程，消息通过消息寄存器从一个线程的 UTCB 块传递到另一个线程的 UTCB 块中。消息传递采用同步方式，传递的数据量跟 IPC 调用的类型有关。Nros 提供三个 ipc 接口，分别为 `ipc_send()`、`ipc_recv()` 和 `ipc_sendrecv()`，下面分别描述这三个接口的实现细节。

1. 消息发送 (`ipc_send`)

下面给出该函数的核心代码，忽略错误处理：

```

int ipc_send(l4id_t recv_tid, unsigned int flags)
{
    struct ktcb *receiver;
    struct waitqueue_head *wqhs, *wqhr;
    int ret = 0;

    if (!(receiver = tcb_find_lock(recv_tid)))
        return -ESRCH;

    wqhs = &receiver->wqh_send;

```

```
wqhr = &receiver->wqh_recv;

// 判断接收者是否已经准备接收
if (receiver->state == TASK_SLEEPING &&
    receiver->waiting_on == wqhr &&
    (receiver->expected_sender == current->tid ||
     receiver->expected_sender == L4_ANYTHREAD)) {
    struct waitqueue *wq = receiver->wq;

    // 从等待队列中移除
    list_remove_init(&wq->task_list);
    wqhr->sleepers--;
    task_unset_wqh(receiver);

    // 拷贝消息
    if ((ret = ipc_msg_copy(receiver, current)) < 0)
        ipc_signal_error(receiver, ret);

    // 异步唤醒接收者
    sched_resume_async(receiver);

    return ret;
}

// 如果没有准备好，将本线程加入等待队列，并睡眠
CREATE_WAITQUEUE_ON_STACK(wq, current);
wqhs->sleepers++;
list_insert_tail(&wq.task_list, &wqhs->task_list);
task_set_wqh(current, wqhs, &wq);
sched_prepare_sleep();

// 调度，运行其他线程
schedule();

return ipc_errors();
```

```
}
```

上面是消息发送的核心代码，逻辑简单直接。主要过程是首先判断接收者是否已经准备好接受发送者的消息，如果是，则将接收者从等待队列移除，并调用 `sched_resume_async()` 异步唤醒接收者，然后发送结束，线程对出内核态，进入用户空间，下次接收者被调度运行的时候，已经可以从自身的 `UTCB` 结构中得到需要的信息了。如果不是，则进入等待队列，睡眠。

2. 消息接收 (`ipc_recv`)

```
int ipc_recv(l4id_t senderid, unsigned int flags)
{
    struct waitqueue_head *wqhs, *wqhr;
    int ret = 0;

    wqhs = &current->wqh_send;
    wqhr = &current->wqh_recv;

    current->expected_sender = senderid;

    // 首先判断等待队列是否为空
    if (wqhs->sleepers > 0) {
        struct waitqueue *wq, *n;
        struct ktcb *sleeper;

        // 线性查找处于等待状态的发送者
        list_foreach_removable_struct(wq, n, &wqhs->task_list, task_list) {
            sleeper = wq->task;

            // 根据线程 ID 来判断发送者
            if ((sleeper->tid == current->expected_sender) ||
                (current->expected_sender == L4_ANYTHREAD)) {
                list_remove_init(&wq->task_list);
                wqhs->sleepers--;
                task_unset_wqh(sleeper);

                if ((ret = ipc_msg_copy(current, sleeper)) < 0)
                    ipc_signal_error(sleeper, ret);
            }
        }
    }
}
```

```

        sched_resume_sync(sleeper);
        return ret;
    }
}

// 如果没有找到，则进入睡眠状态
CREATE_WAITQUEUE_ON_STACK(wq, current);
wqhr->sleepers++;
list_insert_tail(&wq.task_list, &wqhr->task_list);
task_set_wqh(current, wqhr, &wq);
sched_prepare_sleep();

schedule();

return ipc_handle_errors();
}

```

消息接收的逻辑与发送流程相似，不再详细解释。

3. 消息发送和接收（ipc_sendrecv）

该接口使用一次系统调用完成消息的发送和接收，主要用于客户端/服务器模型下的通信。

核心代码如下：

```

int ipc_sendrecv(l4id_t to, l4id_t from, unsigned int flags)
{
    int ret = 0;

    if (to == from) {
        // 发送请求
        if ((ret = ipc_send(to, flags)) < 0)
            return ret;
        // 接收请求
        if ((ret = ipc_rcv(from, flags)) < 0)
            return ret;
    }
}

```

```
    return ret;
}
```

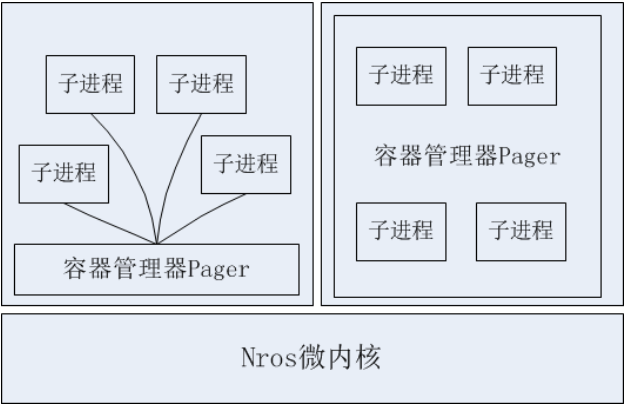
该接口要求发送者线程 ID 和接收者线程 ID 相同，执行过程如下：

- (1) client 线程调用 ipc_sendrecv();
- (2) server 线程使用 ANYTHREAD 参数调用 ipc_rcv(), 接受任何线程的消息;
- (3) 消息接收过程发生，双方交换消息寄存器内容;
- (4,5) client 线程 ipc_send()返回，此后立刻调用 ipc_rcv();
- (4,5) server 线程在用户空间处理 client 请求;
- (6) server 线程处理完请求，调用 ipc_send()将结果发送给 client 线程;
- (7) 消息接收过程发生，client 线程 ipc_rcv()返回，整个过程结束。

4.3 容器

4.3.1 容器概述

容器将系统分割成多个独立的执行空间，每个容器都有一组线程、地址空间、设备和其他资源。容器和虚拟机很相似。容器的数量和物理资源是静态赋予的，在构建这个内核和容器二进制文件的时候就确定了。关于 Nros 的构建过程和二进制形式后面会描述，容器的结构在本章阐述。容器的方式提供了巨大的灵活性，可以实现任意的软件架构。



4.1 两种容器模型

图 4.1 左侧表示将 Pager 实现为一个 Server，Pager 为每个进程创建单独的地址空间，每个进程相当于一个 Client；右侧表示子进程与 Pager 进程共享同一个地址空间，相当于一个应用程序。这两者分别对应两种模型：

1) client/server 模型，Pager 用于创建子线程和地址空间，然后管理这些线程和空间。在 Nros 上构建虚拟操作系统内核即可采用此种方案。

2) 独立应用程序模型，这种情形下，应用程序就是 **Pager** 本身，它在自己的地址空间创建若干线程。

4.3.2 容器的结构

容器与 **Pager** 密切相关，通常每个容器只有一个 **Pager**，用于管理内存空间和管理子线程，但视需要也可以有多个 **Pager**。其结构如下：

```
struct Pager {
    struct ktcb *ktcb;           // Pager 的 ktcb 结构
    unsigned long start_lma;     // 线性地址起始
    unsigned long start_vma;     // 虚拟地址起始
    unsigned long start_address; // 代码起始地址
    unsigned long stack_address; // 栈地址
    unsigned long memsize;       // 内存大小
    struct cap_list cap_list;    // 权能列表

    unsigned long rw_pheader_start;
    unsigned long rw_pheader_end;
    unsigned long rx_pheader_start;
    unsigned long rx_pheader_end;
}
```

容器的结构如下：

```
struct container {
    l4id_t cid;                  /* 容器 ID */
    int nPagers;                 /* Pager 数量 */
    struct link list;            /* 容器链表 */
    struct address_space_list space_list; /* 地址空间链表 */
    char name[CONFIG_CONTAINER_NAMESIZE]; /* 容器的名字 */
    struct ktcb_list ktcb_list;  /* 容器内线程链表 */
    struct link Pager_list;      /* Pager 链表 */

    struct id_pool *thread_id_pool; /* id 持 */
    struct id_pool *space_id_pool;

    struct mutex_queue_head mutex_queue_head; /* 互斥队列 */
}
```



```

    struct cap_list cap_list;                /* 容器的全部权能 */
    struct Pager *Pager;                     /* 启动时的 Pager 数组 */
}

```

4.3.3 容器的静态结构

容器的结构体为：

```

struct container_info {
    char name[CONFIG_CONTAINER_NAMESIZE];
    int nPagers;
    int ncaps;
    struct cap_info caps[CONFIG_MAX_CONT_CAPS];
    struct Pager_info Pager[CONFIG_MAX_PAGERS_USED];
};

```

其中包含容器关联的权能信息数量、权能信息数据和 Pager 信息数量和 Pager 信息数据，权能信息和 Pager 信息的结构体分别为：

```

struct cap_info {
    l4id_t target;
    unsigned int type;
    u32 access;
    unsigned long start;
    unsigned long end;
    unsigned long size;
    unsigned int attr;
    l4id_t irq;
};

struct Pager_info {
    unsigned long Pager_lma;
    unsigned long Pager_vma;
    unsigned long Pager_size;
    unsigned long start_address;
    unsigned long stack_address;
    unsigned long rw_pheader_start;
}

```

```

    unsigned long rw_pheader_end;
    unsigned long rx_pheader_start;
    unsigned long rx_pheader_end;
    int ncaps;
    struct cap_info caps[CONFIG_MAX_PAGER_CAPS];
};

```

这些结构体是由 CML 配置系统在配置的时候生成,并插入到最终生成的 ELF 格式文件中去。所有容器都会生成 ELF 格式的文件,链接器将所有容器以二进制性质直接复制到 final.elf, 并在这里插入这些静态信息这个文件里, 可以使用 QEMU 等虚拟机直接启动内核。在内核启动之初, 系统先根据 ELF 格式, 将各个容器取出, 根据 ELF 的信息将指定的 Section 提取拷贝到内存中去, 然后读取容器的静态信息, 进行初始化。

4.3.4 容器的初始化

Nros 构建产生的二进制格式是 UNIX 平台通用的 ELF 格式, 使用 ELF 格式原因是, 操作 ELF 文件的工具和库比较丰富, 并且可以用通用的调试器调试。系统的加载过程在前面描述过。这里从容器的初始化入口开始描述容器是如何初始化的。

容器的初始化包括容器资源的初始化和 Pager 的初始化。容器初始化是资源初始化的一部分, 相关的参数在构建时确定, 系统启动时读入全局资源表中。资源初始化首先初始化各种基本信息, 如物理内存大小、虚拟内存大小、设备内存地址区域等。

过程如代码所示:

```

for (int i = 0; i < bootres.nconts; i++) {
    container = container_alloc_init();
    copy_container_info(container, &cinfo[i]);
    kres_insert_container(container, kres);
}
container_init_Pagers(kres);

```

启动时会首先判断容器的个数和位置, 然后内核为容器分配内存, 并根据初始化信息填充数据结构, 将容器加入全局资源表中, 最后调用 container_init_Pagers() 初始化各个容器的 Pager。该函数会遍历各个容器, 并针对每个 Pager 进行如下过程的初始化:

- 1) 分配 ktc_b 结构;
- 2) 设置任务的 Pager, container, 线程 ID;
- 3) 创建新的地址空间, 并插入到容器的地址空间链表里;

- 4) 映射 Pager 相关的页;
- 5) 初始化调度器;
- 6) 设置 Pager 的权能;
- 7) 将刚创建的任务插入到可执行队列, 后面运行;
- 8) 将任务加入全局任务链表里。

4.4 内存管理

4.4.1 内存管理概述

与大部分传统内核不同, Nros 不存在物理内存管理, 物理内存完全由用户程序自己控制, 就是说在 Nros 里, 根本不存在“物理内存管理器”这样的东西^[42]。内核不需要“伙伴系统”, 也不需要“slab 分配器”。内核会占用一小部分内存用于自身的运行, 除此以外的物理内存都用于应用程序的运行。前面章节已经介绍, 容器在 Nros 里的是静态配置的, 每个容器所占用的物理内存也是在静态分配的。物理内存的管理, 完全是用户空间的责任, 但是内核会提供虚拟内存的映射机制。通过这种方式, Nros 完全将内存管理抛离内核空间, 放到用户空间。容器的物理内存保护由权能保护, 保证不同容器资源的独立可靠。

现代操作系统使用虚拟内存做进程空间的隔离, 虚拟内存使得所有进程享有相同的地址空间, 使得相同的地址模型成为可能。微内核, 尽管并不直接参与进程模型的构建, 但必须提供构建进程独立地址空间的基础设施。只有这样, 微内核才具有通用性。Nros 支持虚拟内存, 并且在系统启动之初就已经打开了虚拟内存。由于虚拟内存的开启和映射与体系结构密切相关, 下一节将简单描述 ARM 体系结构的虚拟内存机制和内存管理单元(MMU)的技术信息, 然后描述 Nros 是如何处理内存映射和为上层提供映射服务。

4.4.2 ARM 内存管理单元

ARM 体系结构下, 虚拟内存到物理内存的转换是由 MMU (内存管理单元) 完成的。MMU 控制寻表硬件访问内存中的页表, 由内存中的一个两级页表控制 TLB (Translation Lookaside Buffer) 中的指令和数据, 完成后, 虚拟内存到物理内存的转换信息被放置在 TLB 里。ARM 的 MMU 是作为协处理器来实现的, 即协处理器 CP15。ARM 的 MMU 实现了虚拟地址的映射、访问权限和高速缓存的管理。当 CPU 访问内存时, MMU 先查找 TLB 中的虚拟地址表。如果 TLB 没有命中, 转换表会遍历内存中的转换, 获得转换和访问权限。该过程结束后,

结果将被放到 TLB 中，此时即得到访问入口。TLB 中的信息包括：

- 1) 是否使用高速缓冲；
- 2) 访问权限信息；
- 3) 如果 cache 没有命中，则物理地址作为行抓取（line fetch）硬件的输入地址。如果命中，忽略物理地址，数据直接从 cache 里取出。

开启虚拟内存需要将 CP15 的 c1 控制寄存器的 M 位设置为 0。在系统重置（reset）时，该位被清零，默认关闭 MMU。有两个转换表基地址控制寄存器 TTBCR（Translation Table Base Control Register）。第一级描述符表示是访问 section 还是访问页表。如果访问页表，MMU 就抓取第二级描述符。页表包含 256 个 4KB 大小的页表项。可以检查二级页表描述符的最低两位，判断是何种页类型。不论是一级还是二级描述符，只要最低两位都是零，则表示虚拟内存没有被映射，访问这样的地址会产生一次转换错误。高位会被 CPU 忽略，所以内核可以根据自己的目的，自行设置这两个描述符的高 30 位。

虚拟地址被分成两个区域：

0x0 -> 1<<(32-N) 由 TTBR0 控制

1<<(32-N) -> 4GB 由 TTBR1 控制

N 的取值设置在 TTBCR 里，如果 N 是零，TTBR0 用于所有地址，这是 V5 的行为；如果 N 不是零，内核和 IO 内存放置在高地址区域，有 TTBR1 控制，用户任务所占用内存使用低端内存，由 TTBR0 控制。ARM 的 MMU 模型提倡为每个进程维护一个页表，然后在上下文切换的时候修改 TTBR0 寄存器。使用这种页表而使用完全页表，使得整个系统可以使用更少的内存存储页表，不同的进程也可以拥有不同大小的页表。只需要在上下文切换的时候修改 N 的值即可。

4.4.3 虚拟内存模型

Nros 采用了与 Linux 相似的虚拟内存管理模型，即将虚拟内存分成三级页表，该模型与 Intel 体系结构所支持的硬件虚拟内存模型类似，由此可以方便的移植到 x86 或其他具有类似模型的体系结构。

该模型使用的三级页表结构，其相关数据结构如下：

```
typedef u32 pmd_t;
typedef u32 pte_t;

typedef struct pgd_table {
    pmd_t entry[PGD_ENTRY_TOTAL];
} pgd_table_t;
```

```
typedef struct pmd_table {
    pte_t entry[PMD_ENTRY_TOTAL];
} pmd_table_t
```

其中 PGD 表示全局页目录 (Page Global Directory), PMD 表示中级页目录 (Page Middle Directory), PTE 表示页表项 (Page Table Entry)。本内核采用的虚拟内存转换机制是基于 Section 的二级页表结构。

4.4.4 虚拟内存接口

Nros 根据上一节定义的结构, 封装了一层平台无关的接口。下面说明各接口的功能, 并分析重要接口的实现。

```
1) arch_add_section_mapping_init(unsigned int paddr,
                                unsigned int vaddr,
                                unsigned int size,
                                unsigned int flags)
```

该函数映射物理内存 paddr 到虚拟内存 vaddr, 所使用的 section 大小为大于等于 size 的大小。然后初始化这些页表项的标志和地址。核心逻辑如下:

```
/* 得到一级页表地址 */
l1_ptab = virt_to_phys(&init_pgd);
/* 得到 vaddr 里该页表的偏移 */
l1_offset = (vaddr >> 18) & 0x3FFC;
/* 页表项的其实位置 */
ppte = (unsigned int *) (l1_ptab + l1_offset);
for(int i = 0; i < size; i++) {
    *ppte = 0; /* 清空旧值 */
    *ppte |= paddr; /* 写入物理地址 */
    *ppte |= PMD_TYPE_SECTION; /* 设置地址转换类型位 */
    /* 仅允许内核使用 */
    *ppte |= (SVC_RW_USR_NONE << SECTION_AP0);
    *ppte |= flags; /* 权能等标志 */
    ppte++; /* 下一个 section 表项 */
    paddr += SECTION_SIZE; /* 下一个物理地址 section */
}
```

该步骤逻辑简单, 主要是按 ARM 的芯片手册, 循环页表设置各个数据位。

2) void write_pte(pte_t *ptep, pte_t pte, u32 vaddr)

写页表项。

3) void write_pmd(pmd_t *pmd_entry, u32 pmd_phys, u32 vaddr)

写中级页目录项。

4) pte_t virt_to_pte_from_pgd(unsigned long virtual, pgd_table_t *pgd)

从全局页表里得到指定虚拟内存的页表项。

5) unsigned long virt_to_phys_by_pgd(unsigned long vaddr, pgd_table_t *pgd)

从全局页表里得到指定虚拟内存的物理地址。

6) void attach_pmd(pgd_table_t *pgd, pmd_table_t *pmd, unsigned int vaddr)

将 vaddr 所在的中级页目录添加到全局页表里。

7) pgd_table_t *copy_page_tables(pgd_table_t *from)

克隆用户空间页表，用于 fork 进程。该过程结束后，新的页表会共享内核的相关的内存映射，但会有一份私有的用户空间内存映射。核心逻辑如下：

a) 分配全局页目录

b) 拷贝整个 PGD

c) 对于 PGD 内所有的 PMD 表项：

如果不是内核 PMD 项：

拷贝 PMD 内所有 PTE 项

8) void space_switch(struct ktc *to)

切换地址空间，该过程涉及很多特权级指令的操作，主要逻辑如下：

a) pgd_table_t *pgd = TASK_PGD(to); //得到新的 PGD

b) arm_clean_invalidate_cache(); //清空 cache，数据写入内存

c) arm_invalidate_tlb(); //刷新 cache

d) arm_set_ttb(virt_to_phys(pgd)); //写入新的 PGD

e) arm_invalidate_tlb() //刷新 cache

这里需要注意 ARM 的术语，“刷新（flushing）”CACHE 表示使 CACHE 的内容无效，“清空（cleaning）”CACHE 表示将 CACHE 的数据写入内存。在回写型（write-back）的 CACHE 里，必须先清空再刷新。arm_set_ttb 其实是汇编函数，该函数设置 CP15 协处理器，并修改 PC：

```
mcr p15, 0, r0, C15_ttb, c0, 0
```

```
mov    pc, lr
```

9) void copy_pgd_kernel_entries(pgd_table_t *to)

拷贝内核全局页表项。

4.4.5 地址空间 (Address Space)

每个地址空间有一个全局页目录，地址空间是高层接口。一个地址空间可以包含多个线程，每个线程仅属于一个地址空间。地址空间主要由 **Pager** 来创建和删除。**Nros** 使用地址空间来为软件实现提供进程的概念，从概念上讲，一组运行于同一个地址空间的线程可以看作是一个进程。它们之间共享该地址空间内的全局数据。将地址空间的概念留给用户空间使用，这是一种比 **UNIX** 传统的 `fork()+mmap()` 调用更加灵活的方式，使用地址空间加上线程控制的功能，我们可以很容易的实现 `fork()` 调用，甚至 **Linux** 的 `clone()` 调用。在下一节，我们讲用一个例子来说明说明如何使用地址空间和线程控制来实现 `fork` 调用。

下面首先介绍地址空间相关的数据结构，然后介绍相关接口。

```
struct address_space {
    l4id_t spid;           // 地址空间 ID
    struct link list;      // 链表
    struct spinlock lock;  // 自旋锁
    pgd_table_t *pgd;      // 关联的 PGD
    struct cap_list cap_list; // 权能列表
    int ktc_b_refs;        // 线程引用计数
};
```

该数据结构很简单，唯一需要注意的是 `ktc_b_refs`，它是内核中地址空间的引用计数，表示在该地址空间里有多少个线程。

地址空间的接口共有 8 个，下面分别描述：

1) `struct address_space *address_space_create(struct address_space *orig);`

创建新的地址空间。主要逻辑：a) 分配 `address_space` 结构，并初始化；b) 分配 PGD；c) 调用上一节中定义的 `copy_pgd_kernel_entries()` 拷贝内核页表项；d) 分配并设置新地址空间的 `SPID`；e) 如果提供了地址空间，则拷贝整个空间。

2) `void address_space_delete(struct address_space *space, struct cap_list *clist);`
删除地址空间。主要是资源回收。

3) `void address_space_attach(struct ktc_b *tcb, struct address_space *space);`
将 `ktc_b` 的 `space` 字段关联到该地址空间，并增加地址空间的引用计数。

4) `struct address_space *address_space_find(l4id_t spid);`
根据 `SPID` 便利链表，查找地址空间。

5) `void address_space_add(struct address_space *space);`
将地址空间加入全局地址空间链表里。

6) `void address_space_remove(struct address_space *space, struct container`

```
*cont);
```

将地址空间从全局地址空间移除。

```
7) void init_address_space_list(struct address_space_list *space_list);
```

初始化全局地址空间链表。

```
8) int check_access_task(unsigned long vaddr, unsigned long size,
                        unsigned int flags, int page_in, struct ktcb *task);
```

检查给定的虚拟空间地址是否是有效的用户控件地址。如果是，判断当前是否映射在本地址空间中。如果没有映射，给线程的 Pager 发送一条 page-in 的请求。如果页错误状态还没有清空，则异常。事实上，该实现并不能保证内核可以访问用户空间的指针。Pager 可以映射一个内核发起的映射请求，但是在内核已经访问之前取消映射了。如果要解决这个问题，需要为每个 PTE 设置一个锁，并且在 map 调用的时候检查该锁是否被锁住，如果是则无动作。内核需要映射时先锁住该 PTE，以此保证用户程序不会干扰。

4.4.6 地址空间的使用举例（使用地址和线程接口实现 fork()调用）

由于篇幅关系，这里只列出主要逻辑的代码，其他部分省略。

```
int sys_fork(struct ktcb *parent) {
    int err;
    struct ktcb *child;
    struct exregs_data exregs;
    struct task_ids ids;
    struct Pager *Pager;
    struct address_space* space;

    vm_freeze_shadows(parent);           // 冻结父进程，使其只读
    child = task_create(parent, &ids);    // 创建子进程
    space = address_space_create(0);      // 创建新的地址空间
    address_space_add(space);             // 加入全局链表
    Pager = clone_Pager(parent->Pager);  // 克隆 Pager
    address_space_attach(ktcb, space);    // 关联地址空间和任务
    child->Pager = Pager;                 // 关联 Pager

    memset(&exregs, 0, sizeof(exregs));
    exregs_set_mr(&exregs, MR_RETURN, 0);
```



```
    exregs_set_utcb(&exregs, child->utcb_address);

    err = l4_exchange_registers(&exregs, child->tid);
    global_add_task(child);
    l4_thread_control(THREAD_RUN, &ids);
    return child->tid;
}
```

该函数首先冻结父进程的地址空间，防止 fork 的过程中造成修改，然后创建子进程，创建地址空间，拷贝父进程的 Pager，设置关联。exregs 相关的调用用于设置刚刚创建的进程的状态，当状态设置完毕，调用 l4_thread_control 启动进程。通过该函数可以看出 Nros 的地址空间模型的灵活性。系统并不强制要求某种编程风格，相反它只提供一组基本的系统调用和库函数，用户程序根据需实现自己的服务。

第五章 构建与测试

5.1 构建

本节一步步讲解如何下载、安装、配置编译、运行、调试 Nros 所需的工具。本节假定运行环境是 Ubuntu 11.04 及以上。

1. 为源代码和工具分别创建目录

```
% mkdir /opt
% mkdir /opt/archives
% mmdir /opt/tools
```

这里分别为源码和工具创建目录，本节后面的步骤都是用这个目录结构。

2. 安装 Git

分布式版本控制系统 Git 允许在 github.com 上下载和管理 Nros 的源代码。

对于 Ubuntu 用户，在 shell 里输入以下命令安装：

```
% sudo apt-get install git-core
```

从 Ubuntu 仓库安装任何软件都需要输入 `sudo` 密码。对于 Ubuntu 用户，上面的命令即可安装 Git，对于非 Ubuntu 用户，可以从 git 官网找到相关安装步骤信息。

3. 安装 SCons

Nros 使用的软件构建工具是 SCons。SCons 是一个基于 Python 的软件构建工具，它最大的优势是可以使用 Python 脚本操作构建步骤，可以利用 Python 语言的所有特性编写复杂的逻辑。

对于 Ubuntu 用户，输入一下命令：

```
% sudo apt-get install scons
```

这个步骤需要输入 `sudo` 密码。对于非 Ubuntu 用户，可以 Scons 官方网站查看相关安装信息。

4. 安装 GCC 交叉编译器

Nros 使用 Codesourcery ARM 交叉编译器编译代码。其他交叉编译器，比如 crosstools 也可能可以，但是没有经过测试。Nros 只用这个交叉编译工具测试过。

需要注意的是，Codesourcery 工具链有两个类型，一类是 EABI，一类是 GNUEABI。前者用于构建裸机嵌入式程序，比如操作系统内核；后者用于构建应用程序，比如 Linux 上的应用程序。构建 Nros 也与此类似：构建 Nros 微内核时使用 EABI 工具链；构建用户空间程序时使用 GNUEABI 工具链。两个编译器都有自己独立的构建文件。

按以下步骤安装交叉编译工具链：

1) 为工具链单独创建一个目录

```
% mkdir /opt/archives/cc
```

2) 下载最新的 gcc arm compiler toolchain for ARM EABI 交叉编译器工具链 (IA32 GNU/Linux 安装包)，下载地址为：

```
http://www.codesourcery.com/sgpp/lite/arm/portal/subscription3053
```

将会下载一个名为"arm-2012q2-68-arm-none.eabi.bin"的可执行文件。

3) 下载最新的 gcc arm compiler toolchain for GNU-LINUX 交叉编译工具链 (IA32 GNU/Linux 安装包)，下载地址为：

```
http://www.codesourcery.com/sgpp/lite/arm/portal/subscription3057
```

将会下载一个名为"arm-2012q3-67-arm-none-linux-gnueabi.bin"的可执行文件
2012q2 和 2012q3 都是经过测试的版本。

4) 将这两个文件保存到/opt/archives/cc 目录下。

5) 运行一下命令，配置并安装工具链：

```
% cd /opt/archives/cc
```

```
% chmod 777 arm-2012q2-68-arm-none-eabi.bin
```

```
% . arm-2012q2-68-arm-none-eabi.bin -i console
```

这几个命令给 bin 文件加入可执行权限，最后这个命令需要 root 权限。运行后就会开始安装过程，这个过程中会有几个选项，都选择默认即可。如果问安装路径，添加"/opt/tools/cc"。可以到 Codesourery 官方网站查看帮助文档。

6) 对 arm-2012q3-67-arm-none-linux-gnueabi.bin 也使用同样的方式安装。

7) 将可执行文件路径加入到.bashrc 文件中，.bashrc 文件是用户的 bash 启动脚本通常位于/home/name 目录下，会在用户启动终端时候运行：

```
% PATH=$PATH:/opt/tools/cc/bin
```

```
% export PATH
```

这两行命令将工具链的安装路径加入到 PATH 环境变量里。注意修改后的.bashrc 必须在重启终端之后才会生效，如果想在当前运行的终端生效，需要运行命令，假设用户名是 jiao，则需要运行：

```
% source /home/jiao/.bashrc
```

8) 要测试工具链是否安装正确，环境路径是否配置正确，在命令行输入以下命令：

```
% arm-none-eabi-gcc -help
```

该命令会显示 gcc 交叉编译的帮助信息。

5. 下载 Nros 源代码

Nros 的代码仓库使用的是 github.com，这是一个基于 git 的项目托管网站，

要下载 Nros 源代码，运行如下命令：

```
% cd /opt/  
% git clone git://github.com/jxwr/Nros
```

然后 clone 过程开始，将会创建一个名为 Nros 的目录。

6. 构建 Nros 代码

这里只提供最基本的构建命令，详细的过程，需要解释配置文件，由于篇幅原因，不在这里详细解释。要想构建 Nros，需要执行/opt/Nros 的 build.py 脚本。

运行一下命令：

```
% cd /opt/Nros  
% ./build.py -c
```

这个命令会一次性执行配置和构建步骤。如果仅仅需要构建，使用上一次产生的配置，运行"./build.py"即可。

1) 构建开始的时候，会弹出一个类似 Linux 的"make menuconfig"的窗口。问一些问题。此处可以按自己的要求选择，也可以使用默认配置。按"x"键保存配置，按键盘的上下键选择上层菜单或进入子菜单。

2) 保存之后，构建过程就开始了。最终会在对应目录产生一个 elf 格式的镜像文件：比如，如果编译中包含容器，则会产生一个 cont0.elf 文件，路径为/opt/Nros/build/cont0/。

3) 产生好这些 elf 文件后，下一个步骤是将所有 elf 文件组合成一个最终的 elf 文件，名字叫做"final.elf"。目录为/opt/Nros/build/。这个最终的 elf 文件内嵌了其他几个 elf，同时还内嵌了一个 elf 操作库，用于在系统启动的时候抽取这些 elf。

构建的细节可以运行"./build.py -h"查看，除了"-c"选项还有"-f"选项用于根据指定的配置文件构建，"-r"选项用于重置配置为默认配置。如果只需要配置 Nros，运行"./configure.py"。

7. 安装 QEMU

QEMU 是开源虚拟机模拟器。Nros 使用 QEMU/ARM Versatile 平台作为开发平台。最新的 QEMU 需要一个小 patch 才能与 GDB/Insight 很好的工作，尤其是 -s 选项，如果没有这个 patch，在 QEMU 启动时，不会等待 gdb 连接。由于过程繁琐且具实效性，具体的打补丁的方法可以参看网络，这里不再详细描述。安装 QEMU 也可以从源码安装，官方文档里有详细说明。

8. 安装 Insight

Insight 是 GDB 的 GUI 前端，主要专注嵌入式平台跨平台调试。运行 Nros 并不一定需要 Insight，但是使用它，我们可以单步调试、查看 CPU 和内存状态，可以极大的提高开发效率。Insight 可以用于内核开发，也可以用于用户态程序开发。

Insight 与 QEMU 兼容性不好，目前已知 6.8 版本兼容性比较好，推荐使用，其他版本可能没办法正常工作。Insight 需要从源码编译安装。

1) 创建一个目录：

```
% mkdir /opt/archives/insight
```

2) 下载 insight 安装包（例如 insight-6.8.tar.bz2）；

3) 保存到/opt/archives/insight；

4) 运行如下命令解压压缩包：

```
% cd /opt/archives/insight
```

```
% tar -xjvf insight-6.8-1.tar.bz2
```

该命令会将 insight 解压到/opt/archives/insight 目录下。

5) Insight 以来 X11 开发库和 termcap 库，这两个库都需要提前安装，可以使用 apt-get 来安装：

```
% sudo apt-get install libncurses5-dev
```

```
% sudo apt-get install libx11-dev
```

在 Ubuntu 系统里，libncurses 包含 termcap 库。

6) 安装其他依赖：

```
% sudo apt-get install texi2html
```

```
% sudo apt-get install texinfo
```

```
% sudo apt-get install libexpat-dev libexpat1-dev
```

在运行 make 之前，需要删除 gdb/Makefile 中的-Werror 选项。

7) 运行一下命令配置并安装 Insight：

```
% mkdir /opt/archives/insight-build
```

```
% cd /opt/archives/insight-build
```

```
% ../insight-6.8-1/configure --prefix=/opt/tools/insight --target=arm-none-eabi  
--program-prefix=arm-none- --with-expat
```

选项"-target="告诉配置文件，我们要调试的目标平台是"arm-none-eabi"。选项"-program-prefix"是所有生成的可执行文件的前缀。例如 arm-none-gdb 和 arm-none-insight，可以避免跟系统本身的工具冲突。选项"-with-expat"用于 GDB 解析 QEMU 传递过来的目标平台 XML 定义信息。

需要注意的是 Insight 需要 GCC4.4.1 版本以上，其他版本可能无法正常工作。运行以下命令编译安装：

```
% make
```

```
% make install
```

运行完毕后，所有的可执行文件都会被放置在指定前缀的目录下。

8) gdbinit 文件

Insight 是 GDB 的 GUI 前端。GDB 启动时会从当前用户的 HOME 目录下读取".gdbinit"文件，需要将以下命令逐行放到.gdbinit 文件里。

a) target remote localhost:1234

这行命令告诉 GDB 如何连接 QEMU。

b) load final.elf

连接后加载镜像文件。

c) sym final.elf

加载符号信息，用于调试。

d) break break_virtual

在 break_virtual 符号处设置断点，该断点处虚拟内存刚刚开启。从此处开始即可单独调试程序。该指令使得每次 GDB 启动之后都会停在进入虚拟内存的位置。

e) continue

f) stepi

这两行指令告诉 GDB 开始执行，然后执行完 break_virtual 处的指令。执行到这里以后，就可以允许 Insight 加载新的符号表了。

g) sym kernel.elf

这行命令告诉 GDB 使用 kernel.elf 的符号信息，这是 Nros 在开启虚拟内存以后使用的符号信息。

这些步骤完事以后，还需要将 Insight 的可执行文件放到 PATH 环境变量里：

```
% PATH=$PATH:/opt/tools/insight/bin
```

```
% export PATH
```

将这两行文件加入到 HOME 目录下的.bashrc 文件里，然后运行：

```
% source /home/username/.bashrc
```

加载到当前 shell 环境里，注意将 username 修改成自己的用户名。

要测试 insight 是否安装正确，运行如下命令：

```
% cd Nros/build
```

```
% arm-none-insight
```

这时应该弹出 Insight 的 GUI 调试窗口。

5.2 测试运行 Nros

5.2.1 运行 Nros

安装完所有的工具之后，就可以运行 Nros 了。当 Nros 成功构建以后，会产

生一个名为 `final.elf` 的可执行文件。该文件包含了一个 `elf` 格式加载库和内核代码，用户程序也嵌入其中。在终端运行：

```
% /opt/Nros/tools/run-qemu-insight
```

该脚本包含了启动 QEMU 的所有命令，加载 `final.elf`，启动 `Insight` 并运行直到开启虚拟内存。这里会运行上一节介绍 `gdbinit` 中的命令，然后会弹出 `Insight` 调试窗口。

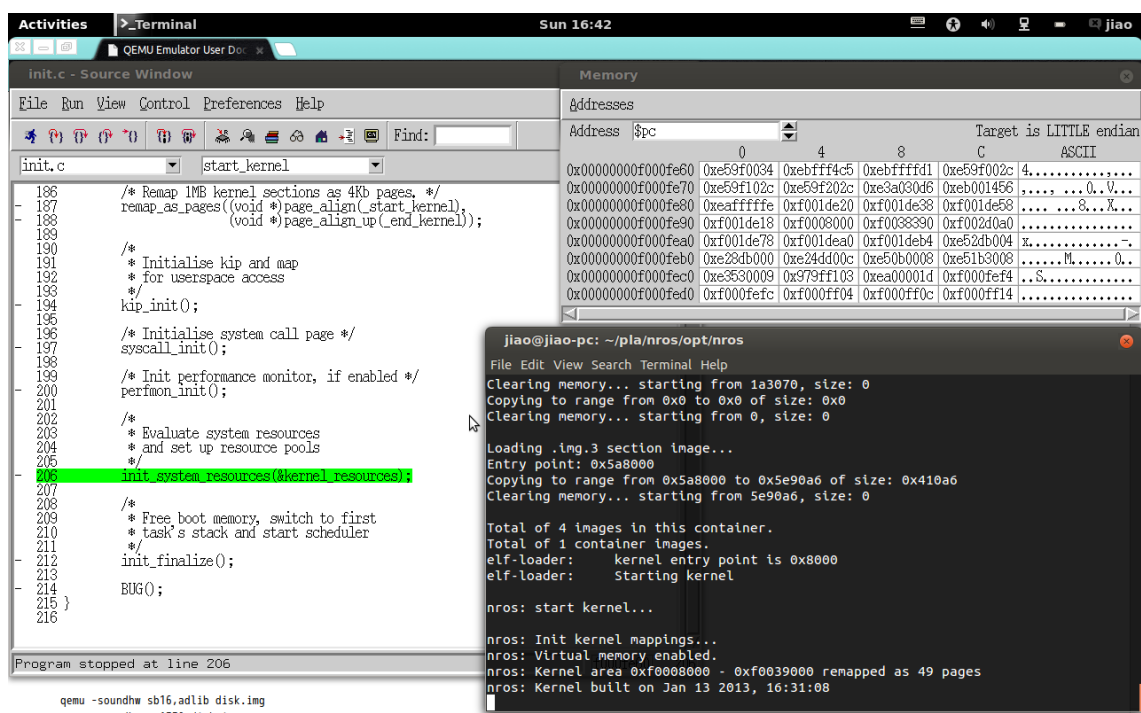
```

111
112     /* Jump to virtual memory addresses */
113     __asm__ __volatile__ (
114         "add    sp, sp, %0 \n" /* Update stack pointer */
115         "add    fp, fp, %0 \n" /* Update frame pointer */
116         /* On the next instruction below, r0 gets
117         * current PC + KOFFSET + 2 instructions after itself. */
118         "add    r0, pc, %0 \n"
119         /* Special symbol that is extracted and included in the loader.
120         * Debuggers can break on it to load the virtual symbol table */
121         "global break_virtual;\n"
122         "break_virtual:\n"
123         "mov     pc, r0          \n" /* (r0 has next instruction) */
124         :
125         : "r" (KERNEL_OFFSET)
126         : "r0"
127     );
128
129     /*
130     * Restore link register (LR) for this function.
131     *
132     * NOTE: LR values are pushed onto the stack at each function call,
133     * which means the restored return values will be physical for all
134     * functions in the call stack except this function. So the caller
135     * of this function must never return but initiate scheduling etc.
136     */
137     __asm__ __volatile__ (
138         "add     %0, %0, %1 \n"
139         "mov     pc, %0          \n"
140         :: "r" (C_builtin_return_address(0)), "r" (KERNEL_OFFSET)
141     );
142
143     /* should never come here */

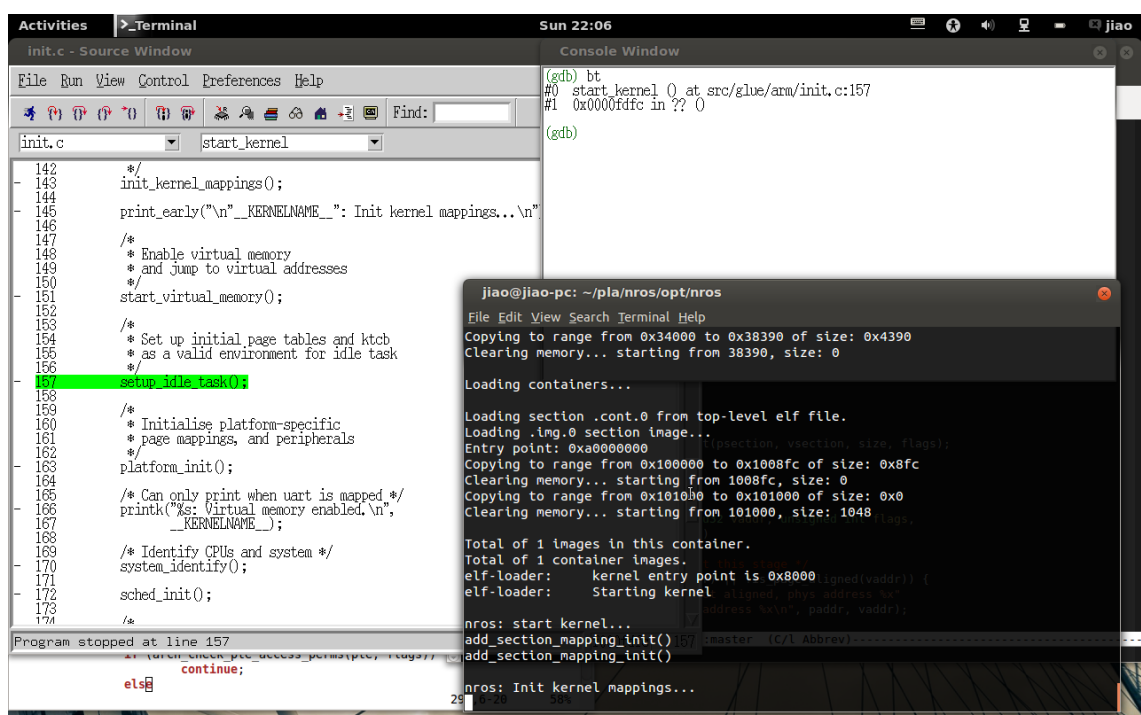
```

5.1 Insight 调试界面

如图 5.1 所示，`Insight` 启动后停在断点 `break_virtual` 处，`break_virtual` 在绿线所在内嵌汇编块前一个块中，与汇编代码书在一起。从此处开始，就可以进行单步运行，使用 `GDB` 和 `Insight` 的全部功能。图 5.2 为查看内存数据，图 5.3 为使用 `GDB` 控制台，使用 `GDB` 脚本调试。



5.2 Nros 调试界面 (1)



5.3 Nros 调试界面 (2)

下面分析一下 run-gemu-insight 中的脚本:

```
% cd /opt/Nros/tools
```

```
% vim run-gemu-insight
```

脚本包含的内容如下:

- a. `cd build`
- b. `qemu-system-arm -s -kernel final.elf -m 128 -M versatilepb -nographic &`
- c. `arm-none-insight; pkill qemu-system-arm`
- d. `cd ..`

下面具体分析脚本内容：

1) 进入 `/opt/Nros/build` 目录；

2) 使用 `qemu-system-arm` 启动 QEMU。"-kernel"选项告诉 QEMU 使用 `final.elf` 镜像文件，该文件在当前目录下；"-m 128"告诉虚拟机我们要使用 128MB 的内存；"-M"告诉虚拟机模拟的平台类型，这里是"ARM Versatile PB"。选项"-s"告诉 QEMU，在开始运行程序之前，等待调试器连接。新版本的 QEMU 这个选项是坏的。选项"-nographic"重定向 PL001UART 到控制台。

3) "arm-none-insight"启动 insight 调试器。

4) 下一个命令"`pkill qemu-system-arm`"只有当调试器结束的时候才会运行，该命令立刻终止 QEMU。

5) 最后回到 `/opt/Nros` 目录。

5.2.2 测试 Nros 实例

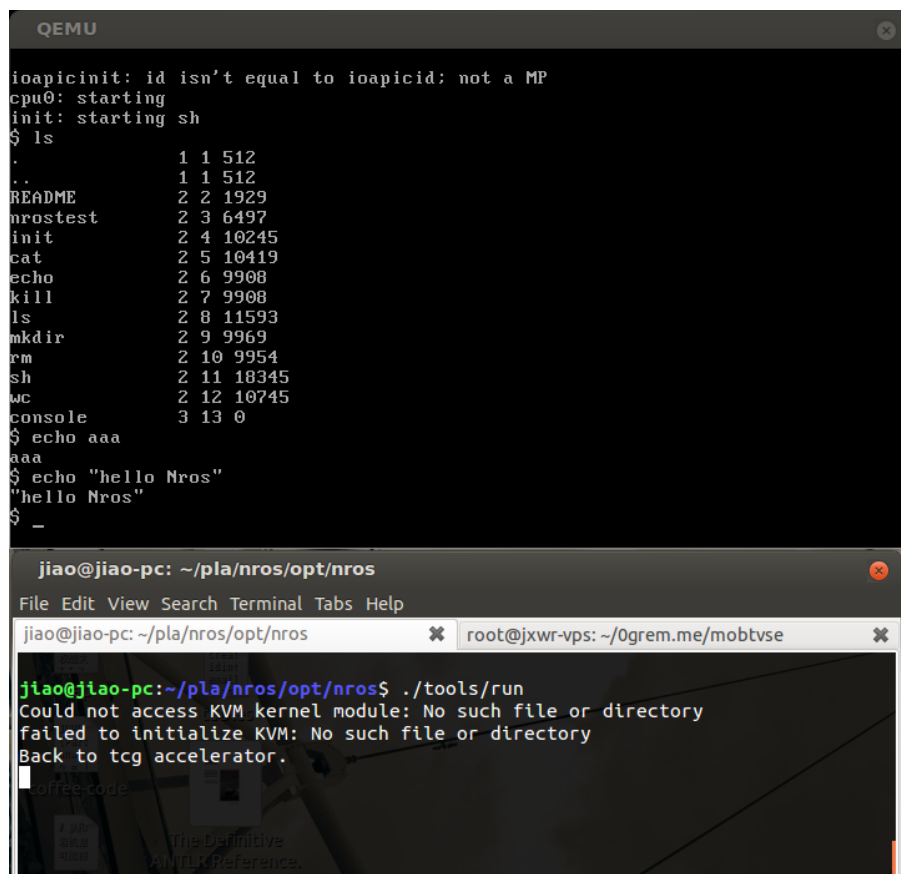
如果不启动 Insight，直接运行 `qemu` 运行 Nros，并使用 Qemu 默认选项开启图形接口，就会弹出一个简陋的 Nros 执行环境，该环境用于测试 Nros。其中包含一个简单的 shell 和几个简单的命令 (`ls`, `echo`, `mkdir`, `wc` 等)。图 5.4 为运行 `ls` 和运行 `echo` 时的系统截图，图 5.5 为运行 `wc` 和 `mkdir` 的系统截图。

Nros 设计了几个例子，这几个例子主要用于演示 Nros 如何工作，也用于从这几个例子开始创建新项目。所有的例子都在 `Nros/config/cml/baremetal` 文件夹下，可以使用以下命令构建指定的例子：

```
% cd codezero
```

```
% ./build.py -f </path/to/config_file>
```

例子的源码都在 `Nros/conts/baremetal` 文件夹下。

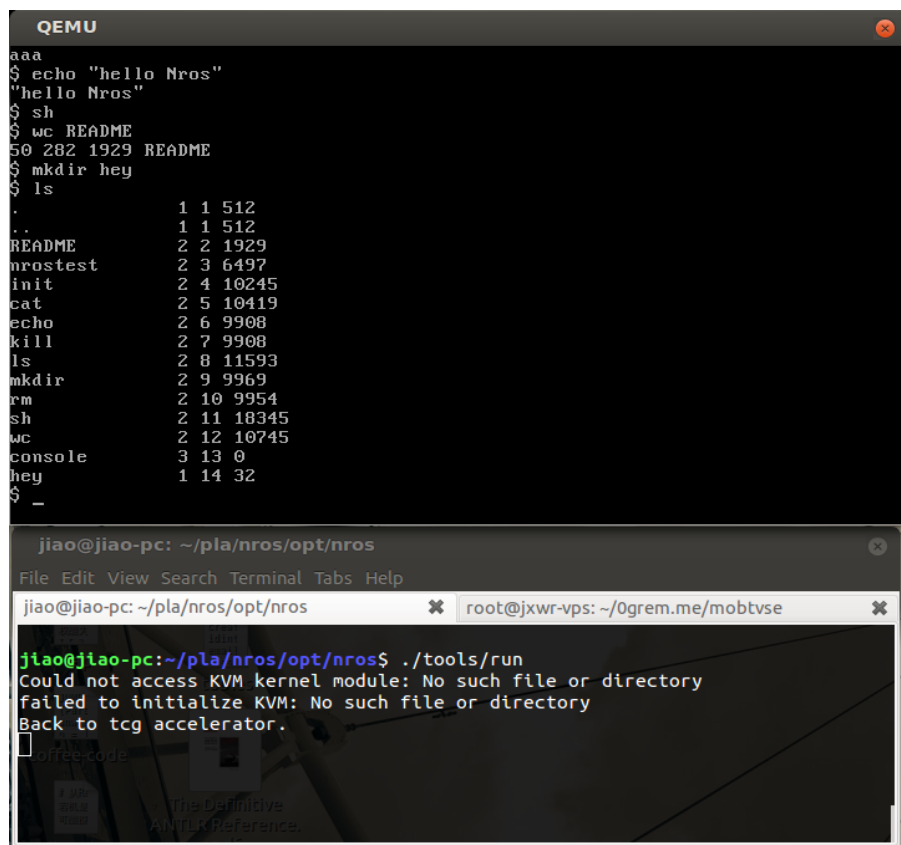


The image shows two overlapping terminal windows. The top window is a QEMU virtual machine terminal. It displays the output of the 'ls' command, showing a directory listing with permissions, owner, group, size, and file names. The bottom window is a host terminal window titled 'jiao@jiao-pc: ~/pla/nros/opt/nros'. It shows the execution of './tools/run', which results in an error message: 'Could not access KVM kernel module: No such file or directory' and 'failed to initialize KVM: No such file or directory', followed by 'Back to tcg accelerator.'

```
QEMU
ioapicinit: id isn't equal to ioapicid; not a MP
cpu0: starting
init: starting sh
$ ls
.          1 1 512
..         1 1 512
README    2 2 1929
nroctest  2 3 6497
init       2 4 10245
cat        2 5 10419
echo       2 6 9908
kill       2 7 9908
ls         2 8 11593
mkdir      2 9 9969
rm         2 10 9954
sh         2 11 18345
wc         2 12 10745
console    3 13 0
$ echo aaa
aaa
$ echo "hello Nros"
"hello Nros"
$ _

jiao@jiao-pc: ~/pla/nros/opt/nros
File Edit View Search Terminal Tabs Help
jiao@jiao-pc: ~/pla/nros/opt/nros  x root@jxwr-vps: ~/0grem.me/mobtvse  x
jiao@jiao-pc:~/pla/nros/opt/nros$ ./tools/run
Could not access KVM kernel module: No such file or directory
failed to initialize KVM: No such file or directory
Back to tcg accelerator.
```

5.4 执行命令 ls 和 echo



The image shows two overlapping terminal windows. The top window is a QEMU virtual machine terminal. It displays the output of the 'wc README' command, showing the word count, line count, and byte count for the README file. The bottom window is a host terminal window titled 'jiao@jiao-pc: ~/pla/nros/opt/nros'. It shows the execution of './tools/run', which results in an error message: 'Could not access KVM kernel module: No such file or directory' and 'failed to initialize KVM: No such file or directory', followed by 'Back to tcg accelerator.'

```
QEMU
aaa
$ echo "hello Nros"
"hello Nros"
$ sh
$ wc README
50 282 1929 README
$ mkdir hey
$ ls
.          1 1 512
..         1 1 512
README    2 2 1929
nroctest  2 3 6497
init       2 4 10245
cat        2 5 10419
echo       2 6 9908
kill       2 7 9908
ls         2 8 11593
mkdir      2 9 9969
rm         2 10 9954
sh         2 11 18345
wc         2 12 10745
console    3 13 0
hey        1 14 32
$ _

jiao@jiao-pc: ~/pla/nros/opt/nros
File Edit View Search Terminal Tabs Help
jiao@jiao-pc: ~/pla/nros/opt/nros  x root@jxwr-vps: ~/0grem.me/mobtvse  x
jiao@jiao-pc:~/pla/nros/opt/nros$ ./tools/run
Could not access KVM kernel module: No such file or directory
failed to initialize KVM: No such file or directory
Back to tcg accelerator.
```

5.5 执行命令 wc 和 mkdir

5.2.3 开发新容器

开发者可以依照如下步骤，自己构建和集成新的容器到内容 Nros 系统。

1) 进入 Nros/conts/baremetal/empty 目录，该目录包含一个空的工程。该工程不做任何事，只是一个项目模板。可以从这里作为起点，开发自己的容器。

2) 在新容器里开发并测试项目。执行构建后，你会看到构建目录创建了 Nros/conts/empty0 容器，所有的源文件都从 Nros/conts/baremetal/empty 拷贝到了 Nros/conts/empty0。

3) 重命名修改后的 empty0，该名字会作为容器的名字，被内核使用。

4) 当项目开发完成之后，就可以将该项目集成到 Nros 构建系统里。然后下次构建时，在配置菜单里就会有该容器的名字出现。添加容器使用如下脚本：

```
% cd Nros
% ./script/baremetal/baremetal_add_container.py -a -i <description> -s
<source_path>
```

这里<description>是该容器的简短描述，<source_path>是容器的源代码目录。如果要删除，运行下面的命令：

```
% ./script/baremetal/baremetal_add_container.py -d -s <source_path>
```

第六章 总结与后续工作

6.1 总结

本文阐述了 L4 微内核理论和 L4 规范的主要接口，通过自己实现的微内核 Nros，以模块为单位，详细阐述了 L4 微内核各核心概念和组件的实现方式。使用容器和权能的概念将系统分割成各个独立的执行环境，同时支持多种软件架构模型。该内核实现了完整的线程模型、线程间通信机制、内存管理模型以及基于权能控制的容器模型。内核的完整代码不到 10000 行，各个模块划分清晰，目前仅支持 ARM 体系结构。

本论文是对 L4 微内核理论实现方式的一次尝试，在此过程中，有许多经验值得总结：

1、实现操作系统并不是件容易的事，尤其是涉及到虚拟内存和硬件设备，然而 L4 微内核使得实现一个操作系统变得非常清晰，使得整个实现过程快速直接。

2、微内核的设计非常强调模块化，如果实现之处不考虑模块化，很多机制的设计会非常被动，甚至需要重写。Nros 的实现过程中，经历数次模块的重写。但是一旦设计完成，实现就是一个轻松愉快的经历。

3、尽量使用尖端的工具辅助开发。目前虚拟机（如 QEMU）非常强大，内核调试工具也已经很完善，为内核编写提供了强有力的支持。如果没有这些工具，很难想象内核开发如何高效的进行。

4、参考成熟的实现方式。在实现 Nros 过程中，作者参考了很多开源 L4 微内核的实现，有些是已经成熟的系统比如 Pistachio 和 seL4，也有些是已经不再维护的内核，比如 L4/MIPS 和 Hazelnut^{[40][48][51]}。前者代码量相对大一些，但融合当前 L4 研究的最新成果，后者相对旧，但是简单易懂且体现了 L4 的精髓。

6.2 后续工作

6.2.1 虚拟化 Linux

Nros 在设计之初曾将内核虚拟化考虑在内，希望可以在该内核基础之上实现 Linux 的虚拟化，可以在用户空间同时运行多个 Linux 实例且互不干扰。然而由于时间所限，并未达到这个目的。

按最初的设想，在 Nros 基础之上实现一个简单用户态和内核态的切换机制，使得不需要修改很多 Linux 内核代码即可实现半虚拟化（Para-Vritralization）。当

下所有基于 L4 的半虚拟化机制基本都是将 Linux 的系统调用和中断处理函数修改成 L4 线程，在发生系统调用和中断时，通过 IPC 的方式将系统调用和中断信息传递到这些 L4 处理线程。Nros 设想的一种机制是，不需要做这样的修改，而仅仅是在 Linux 进入内核态的瞬间，即将 Linux 重新返回到用户态，内核代码在用户态执行。每个 Linux 进程对应一个 Nros 的地址空间，Linux 系统本身作为一个 Pager 存在，负责创建其他进程，并进行内存管理。系统进入内核态后，全局页表并不需要修改，内核的内存位置一直不变。问题是：如何切换到用户态，但继续执行内核代码。执行内核代码本身并没有问题，问题是 CPU 特权级指令如何执行，以及内核页如何防止用户程序访问。C 语言编译成的汇编指令，绝对不会有特权指令，即是说绝大部分的代码根本不需要在内核态运行。需要做的似乎是进入内核态后，马上跳回用户态，然后在用户态执行其他操作，当执行完后，再次进入内核态，完成操作，上下文切换，再次跳转回用户态。这其中有一次上下文切换，会有一定性能损害。需要做的工作是找到所有进入内核态的入口，例如中断、系统调用。以上过程是在设计 Nros 之处的设想，由于时间所限，并未付诸实现。

6.2.2 兼容 POSIX

POSIX 标准非常复杂，且版本很多，不同的实现有不同的语义，且有许多语义不详的角落^[30]。实现一个完整的 POSIX 兼容层并不容易。Nros 只实现了几个基本函数，用于实验。该部分尚需许多工作。

致谢

本文是在鱼滨老师的细心指导下完成的。从文论的立题到最终完成，鱼老师都给予了极大的关怀和帮助，他牺牲了无数宝贵的时间与我探讨论文中涉及到的难题，提出许多有帮助的建议和解决方案。论文的完成是离不开老师的帮助的，在此向老师表示衷心的感谢。

编写系统级的软件是一次历险，每次历险都可以迅速提高一个人的技术水平。在此我要感谢在这次历险过程中陪伴我的同学和师兄师姐们，他们给了我很多有建设性的意见和建议，没有他们陪我一起讨论问题，很多问题不会那么及时得到答案，论文也无法及时达到完成的状态的。

再次感谢在完成此次论文过程中帮助过我的所有人。

参考文献

[1] J. Liedtke, "Towards real microkernels," *Communications of the ACM*, vol. 30, no. 9, pp. 70 – 77, 1996.

[2] J. Liedtke, "On μ -kernel construction," in *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pp. 237 – 250, Copper Mountain, Colorado, USA, December 1995.

[3] L4Ka Team, "L4Ka: pistachio kernel," <http://l4ka.org/projects/pistachio/>.

[4] P.S. Langston, "Report on the workshop on micro-kernels and other kernel architectures," April 1992 <http://www.langston.com/Papers/uk.pdf>.

[5] National ICT Australia, NICTA L4-embedded kernel reference manual version N1, October 2005

<http://ertos.nicta.com.au/software/kenge/pistachio/latest/refman.pdf>.

[6] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid, "UNIX as an application program," in *Proceedings of the Summer USENIX*, pp. 87 – 59, Anaheim, California, USA, June 1990.

[7] D. R. Engler, M. F. Kaashoek, and J. O' Toole Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pp. 251 – 266, Copper Mountain, Colorado, USA, December 1995.

[8] I. M. Leslie, D. McAuley, R. Black, et al., "The design and implementation of an operating system to support distributed multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280 – 1296, 1996. Sergio Ruocco 13

[9] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS '95)*, pp. 78 – 85, Orcas Island, Washington, USA, May 1995.

[10] J. Liedtke, "A persistent system in real use experience of the first 13 years," in *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOS '93)*

pp. 2 – 11, Asheville, North Carolina, USA, December 1993.

[11] D. Hildebrand, "An architectural overview of QNX," in *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 113 – 126, Berkeley, California, USA, 1992.

- [12] B. Leslie, P. Chubb, N. Fitzroy-Dale, et al., “User-level device drivers: achieved performance,” *Journal of Computer Science and Technology*, vol. 20, no. 5, pp. 654 – 664, 2005.
- [13] T. P. Baker, A. Wang, and M. J. Stanovich, “Fitting Linux device drivers into an analyzable scheduling framework,” in *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, July 2007.
- [14] D. Engler, *The Exokernel operating system architecture*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1999.
- [15] J. Kamada, M. Yuhara, and E. Ono, “User-level real time scheduler exploiting kernel-level fixed priority scheduler,” in *Proceedings of the International Symposium on Multimedia Systems*, Yokohama, Japan, March 1996.
- [16] J. Liedtke, “ μ -kernels must and can be small,” in *Proceedings of the 15th International Workshop on Object Orientation in Operating Systems (IWOOOS ’96)*, pp. 152 – 161, Seattle, Washington, USA, October 1996.
- [17] J. Liedtke, “Improving IPC by kernel design,” in *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP ’03)*, pp. 175 – 188, Asheville, North Carolina, USA, December 2003.
- [18] G. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser, “Itanium—a system implementor’s tale,” in *Proceedings of the USENIX Annual Technical Conference (ATEC ’05)*, pp. 265 – 278, Anaheim, California, USA, April 2005.
- [19] L4 headquarters, “L4 kernel reference manuals,” June 2007, <http://l4hq.org/docs/manuals/>.
- [20] G. Klein, M. Norrish, K. Elphinstone, and G. Heiser, “Verifying a high-performance micro-kernel,” in *Proceedings of the 7th Annual High-Confidence Software and Systems Conference*, Baltimore, Maryland, USA, May 2007.
- [21] R. Kaiser and S. Wagner, “Evolution of the PikeOS microkernel,” in *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems (MIKES ’07)*, Sydney, Australia, March 2007.
- [22] H. Härtig and M. Roitzsch, “Ten years of research on L4-based real-time systems,” in *Proceedings of the 8th Real-Time Linux Workshop*, Lanzhou, China, 2006.
- [23] M. Hohmuth, “The Fiasco kernel: requirements definition,” Tech. Rep. TUD-FI98-12, Technische Universität Dresden, Dresden, Germany, December 1998.

[24] M. Hohmuth and H. Härtig, “Pragmatic nonblocking synchronization for real-time systems,” in Proceedings of the USENIX Annual Technical Conference, pp. 217 – 230, Boston, Massachusetts, USA, June 2001.

[25] H. Härtig, R. Baumgartl, M. Borriß, et al., “DROPS: OS support for distributed multimedia applications,” in Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications, pp. 203 – 209, Sintra, Portugal, September 1998.

[26] C. J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig, “Quality-assuring scheduling — using stochastic behavior to improve resource utilization,” in Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS ’01), pp. 119 – 128, London, UK, December 2001.

[27] F. Mehnert, M. Hohmuth, and H. Härtig, “Cost and benefit of separate address spaces in real-time operating systems,” in Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS ’02), pp. 124 – 133, Austin, Texas, USA, December, 2002.

[28] U. Steinberg, J. Wolter, and H. Härtig, “Fast component interaction for real-time systems,” in Proceedings of the EuroMicro Conference on Real-Time Systems (ECRTS ’05), pp. 89 – 97, Palma de Mallorca, Spain, July 2005.

[29] B. Leslie, C. van Schaik, and G. Heiser, “Wombat: a portable user-mode Linux for embedded systems,” in Proceedings of the 6th Linux Conference, Canberra, Australia, April 2005.

[30] ISO/IEC, The POSIX 1003.1 Standard, 1996.

[31] J. S. Shapiro, “Vulnerabilities in synchronous IPC designs,” in Proceedings of the IEEE Symposium on Security and Privacy, pp. 251 – 262, Oakland, California, USA, May 2003.

[32] S. Siddha, V. Pallipadi, and A. VanDeVen, “Getting maximum mileage out of tickless,” in Proceedings of the Ottawa Linux Symposium, Ottawa, Canada, June 2007.

[33] D. Tsafir, Y. Etsion, and D. G. Feitelson, “Secretly monopolizing the CPU without superuser privileges,” in Proceedings of the 16th USENIX Security Symposium, Boston, Massachusetts, USA, April 2007.

[34] Mike Jones, “What really happened on Mars Rover Pathfinder,” The Risks Digest, vol. 19, no. 49, 1997, based on David Wilner’s keynote address of 18th IEEE Real-Time Systems Symposium (RTSS ’97), December, 1997, San Francisco, California, USA. [http://research.microsoft.com/~mbj/Mars Pathfinder/](http://research.microsoft.com/~mbj/Mars%20Pathfinder/).

- [35] J. W. S. Liu, *Real-Time Systems*, Prentice-Hall, Upper Saddle River, New Jersey, USA, 2000.
- [36] V. Yodaiken, “Against priority inheritance,” <http://www.yodaiken.com/papers/inherit.pdf>.
- [37] K. Elphinstone, “Resources and priorities,” in *Proceedings of the 2nd Workshop on Microkernels and Microkernel-Based Systems*, K. Elphinstone, Ed., Lake Louise, Alta, Canada, October 2001.
- [38] D. Greenaway, “From ‘real fast’ to real-time: quantifying the effects of scheduling on IPC performance,” B.Sc.thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2007.
- [39] J. Stoess, “Towards effective user-controlled scheduling for microkernel-based systems,” *Operating Systems Review*, vol. 41, no. 4, pp. 59 – 68, 2007.
- [40] G. Heiser, “Inside L4/MIPS anatomy of a high-performance microkernel,” Tech. Rep. 2052, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
- [41] QNX Neutrino IPC, http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sysarch/kernel.html#NT_OIPC.
- [42] A. Nourai, “A physically-addressed L4 kernel,” B. Eng. thesis, School of Computer Science & Engineering, The University of New South Wales, Sydney, Australia, 2005, <http://www.disy.cse.unsw.edu.au/>.
- [43] B. Ford and S. Susarla, “CPU inheritance scheduling,” in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI ’96)*, pp. 91 – 105, Seattle, Washington, USA, October 1996. 14 *EURASIP Journal on Embedded Systems*
- [44] J. Liedtke, “A short note on cheap fine-grained time measurement,” *Operating Systems Review*, vol. 30, no. 2, pp. 92 – 94, 1996.
- [45] K. Elphinstone, D. Greenaway, and S. Ruocco, “Lazy queueing and direct process switch—merit or myths?” in *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, July 2007.
- [46] S. Ruocco, “Real-time programming and L4 microkernels,” in *Proceedings of the 2nd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.
- [47] “Xenomai—implementing a RTOS emulation framework on GNU/Linux,”

2004, <http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf>.

[48] M. Masmano, I. Ripoll, and C. Crespo, “An overview of the XtratuM nanokernel,” in Proceedings of the Workshop on Operating System Platforms for Embedded Real-Time Applications, Palma de Mallorca, Spain, July 2005.

[49] Politecnico di Milano—Dipartimento di Ingegneria Aerospaziale, “RTAI the real-time application interface for Linux,” <http://www.rtai.org/>.

[50] V. Yodaiken and M. Barabanov, “A real-time Linux,” <http://citeseer.ist.psu.edu/article/yodaiken97realtime.html>.

[51] Iguana OS, <http://www.ertos.nicta.com.au/iguana/>.

[52] S. Ruocco, “User-level fine-grained adaptive real-time scheduling via temporal reflection,” in Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS ’06), pp. 246 – 256, Rio de Janeiro, Brazil, December 2006.

[53] GerHot Heiser and Ben Leslie, The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors